VI-HPS

TAU PERFORMANCE SYSTEM

Sameer Shende
Kevin Huck, Wyatt Spear, Scott Biersdorff
Performance Research Lab

Allen D. Malony, Nick Chaimov, David Poliakoff, David Ozog
Department of Computer and Information Science
University of Oregon

John Linford
ParaTools, Inc.

- Tuning and Analysis Utilities (18+ year project)
- Comprehensive performance profiling and tracing
  - Integrated, scalable, flexible, portable
  - Targets all parallel programming/execution paradigms
- Integrated performance toolkit
  - Instrumentation, measurement, analysis, visualization
  - Widely-ported performance profiling / tracing system
  - Performance data management and data mining
  - Open source (BSD-style license)
- Easy to integrate in application frameworks

http://tau.uoregon.edu

What is TAU?

- TAU is a performance evaluation tool
- It supports parallel profiling and tracing
- Profiling shows you how much (total) time was spent in each routine
- Tracing shows you *when* the events take place in each process along a timeline
- Profiling and tracing can measure time as well as hardware performance counters (cache misses, instructions) from your CPU
- TAU can automatically instrument your source code using a package called PDT for routines, loops, I/O, memory, phases, etc.
- TAU runs on most HPC platforms and it is free (BSD style license)
- TAU has instrumentation, measurement and analysis tools
  - paraprof is TAU's 3D profile browser
- To use TAU's automatic source instrumentation, you may set a couple of environment variables and substitute the name of your compiler with a TAU shell script

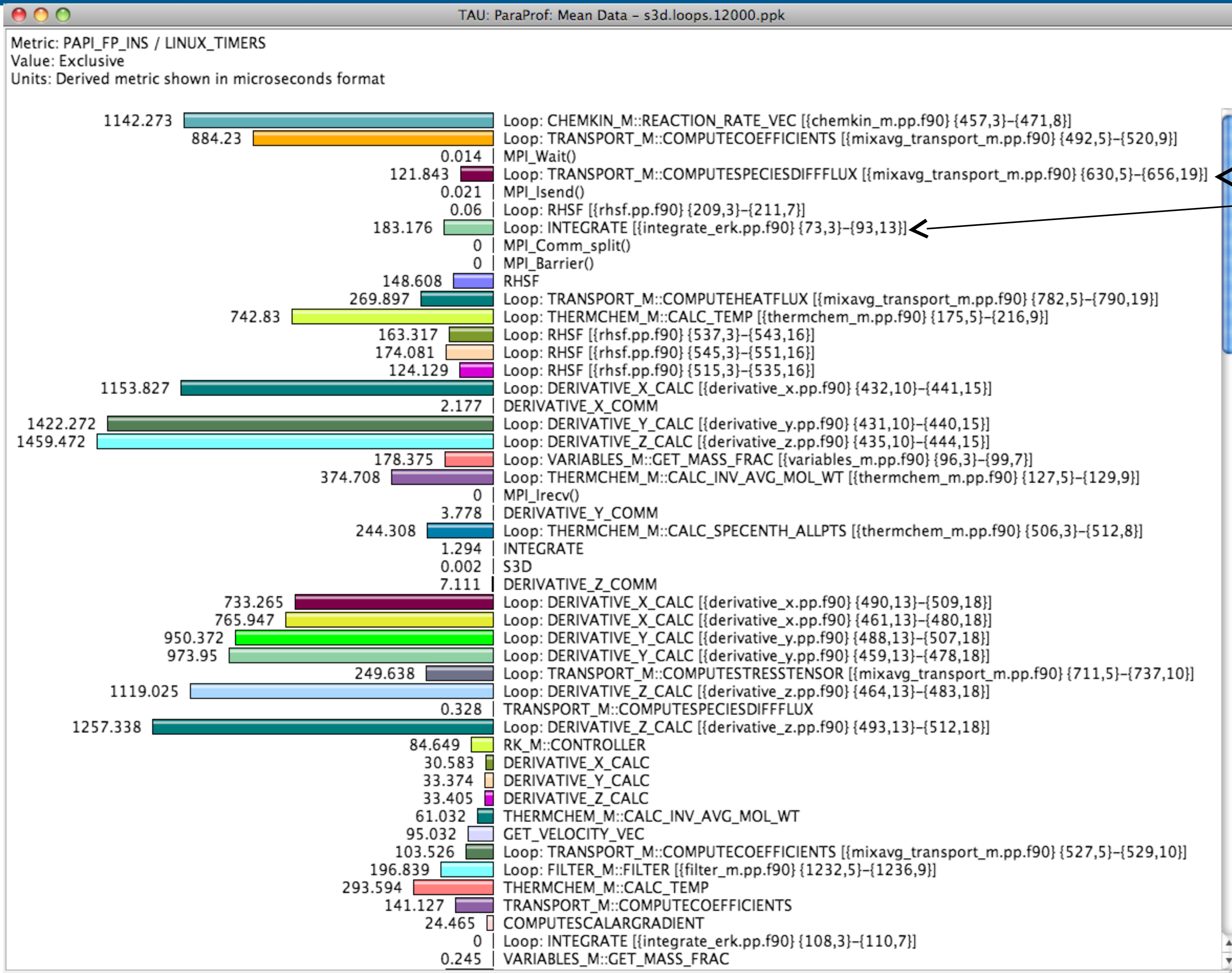
TAU: Usage Scenarios

- How much time is spent in each application routine and outer ***loops***? Within loops, what is the contribution of each ***statement***?
- How many instructions are executed in these code regions? Floating point, Level 1 and 2 ***data cache misses***, hits, branches taken?
- What is the ***peak heap memory*** usage of the code? When and where is memory allocated/de-allocated? Are there any memory leaks?
- How much time does the application spend performing ***I/O***? What is the peak read and write ***bandwidth*** of individual calls, total volume?
- What is the contribution of different ***phases*** of the program? What is the time wasted/spent waiting for collectives, and I/O operations in Initialization, Computation, I/O phases?
- How does the application ***scale***? What is the efficiency, runtime breakdown of performance across different core counts?

Using TAU: Simplest Case

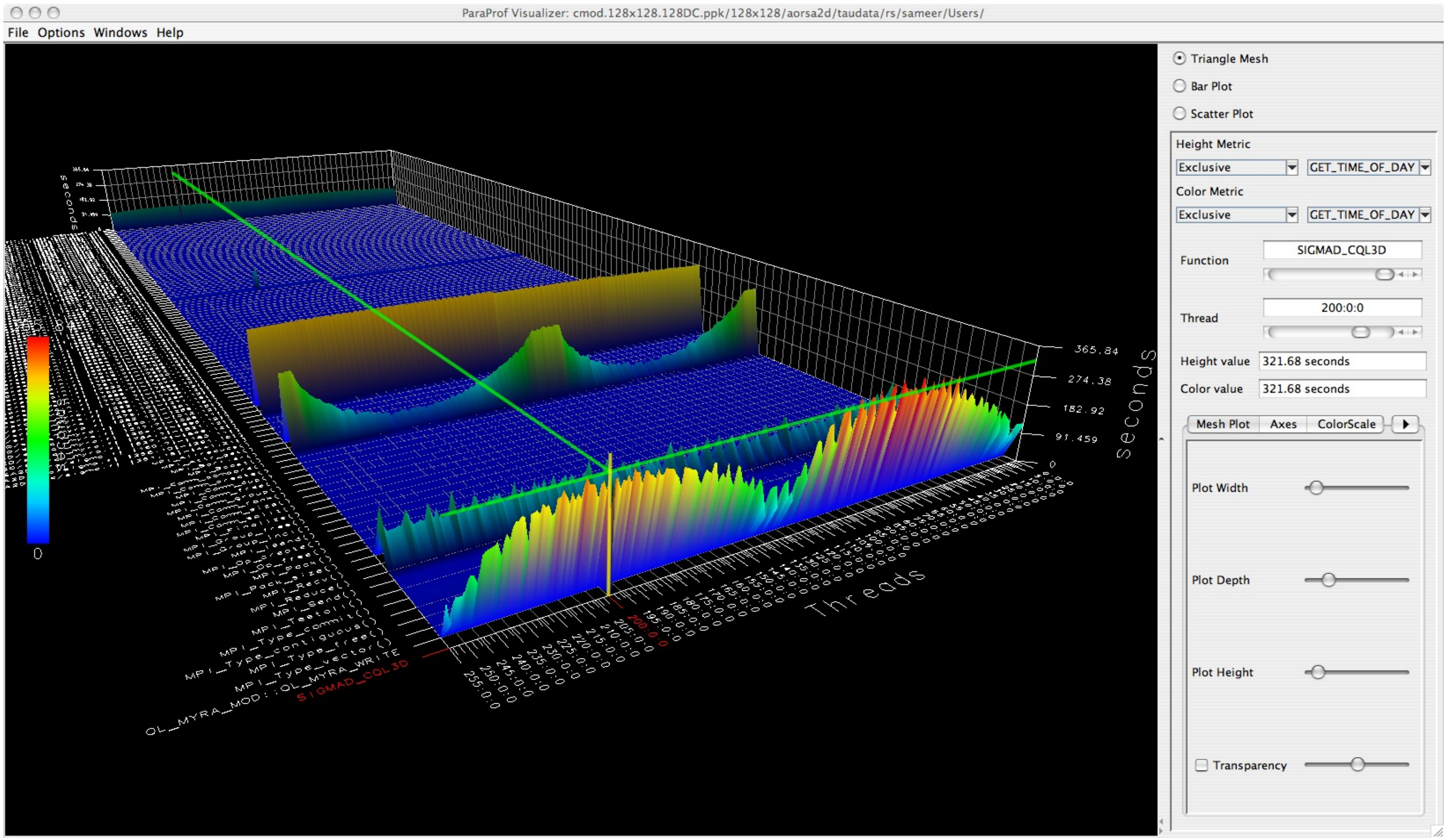
- Uninstrumented code:
  - % mpirun –np 8 ./a.out
- With TAU:
  - % mpirun –np 8 tau_exec ./a.out
  - % paraprof

ParaProf: Mflops Sorted by Exclusive Time



low mflops in loops?

Parallel Profile Visualization: ParaProf



How does TAU work?

- **Instrumentation**: Adds probes to perform measurements
  - Source code instrumentation using pre-processors and compiler scripts
  - Wrapping external libraries (I/O, MPI, Memory, CUDA, OpenCL, pthread)
  - Rewriting the binary executable
- **Measurement**: Profiling or Tracing using wallclock time or hardware counters
  - Direct instrumentation (Interval events measure exclusive or inclusive duration)
  - Indirect instrumentation (Sampling measures statement level contribution)
  - Throttling and runtime control of low-level events that execute frequently
  - Per-thread storage of performance data
  - Interface with external packages (Scalasca, VampirTrace, Score-P, PAPI)
- **Analysis**: Visualization of profiles and traces
  - 3D visualization of profile data in paraprof, perfexplorer tools
  - Trace conversion & display in external visualizers (Vampir, Jumpshot, ParaVer)

Using TAU: A Brief Introduction

- TAU supports several measurement and thread options
  - Phase profiling, profiling with hardware counters, trace with Score-P…
- Each measurement configuration of TAU corresponds to a unique stub makefile and library that is generated when you configure it
- To instrument source code automatically using PDT
  - Choose an appropriate TAU stub makefile in <arch>/lib:

  % export TAU_MAKEFILE=$TAU/Makefile.tau-mpi-pdt

  % export TAU_OPTIONS='-optVerbose …' (see tau_compiler.sh)

  Use tau_f90.sh, tau_cxx.sh or tau_cc.sh as F90, C++ or C compilers:

  % mpif90 foo.f90 **changes to**

  % tau_f90.sh foo.f90
- Set runtime environment variables, execute application and analyze performance data:

  % pprof (for text based profile display)

  % paraprof (for GUI)

Choosing an Appropriate TAU_MAKEFILE

% cd $TAUROOTDIR/<arch>/lib; ls Makefile.*
Makefile.tau-pdt
Makefile.tau-mpi-pdt
Makefile.tau-pthread-pdt
Makefile.tau-*papi*-mpi-pdt
Makefile.tau-mpi-pthread-pdt
Makefile.tau-papi-pthread-pdt
Makefile.tau-opari-openmp-mpi-pdt
Makefile.tau-papi-mpi-pdt-epilog-*scalasca*-trace
Makefile.tau-papi-mpi-pdt-*vampirtrace*-trace …

- For an MPI+F90 application, you may choose **Makefile.tau-mpi-pdt**
  - Supports MPI instrumentation & PDT for automatic source instrumentation
  - % export TAU_MAKEFILE=$TAU/Makefile.tau-mpi-pdt
  - % tau_f90.sh matrix.f90 -o matrix
  - % mpirun –np 8 ./matrix
  - % paraprof

TAU Instrumentation Approach

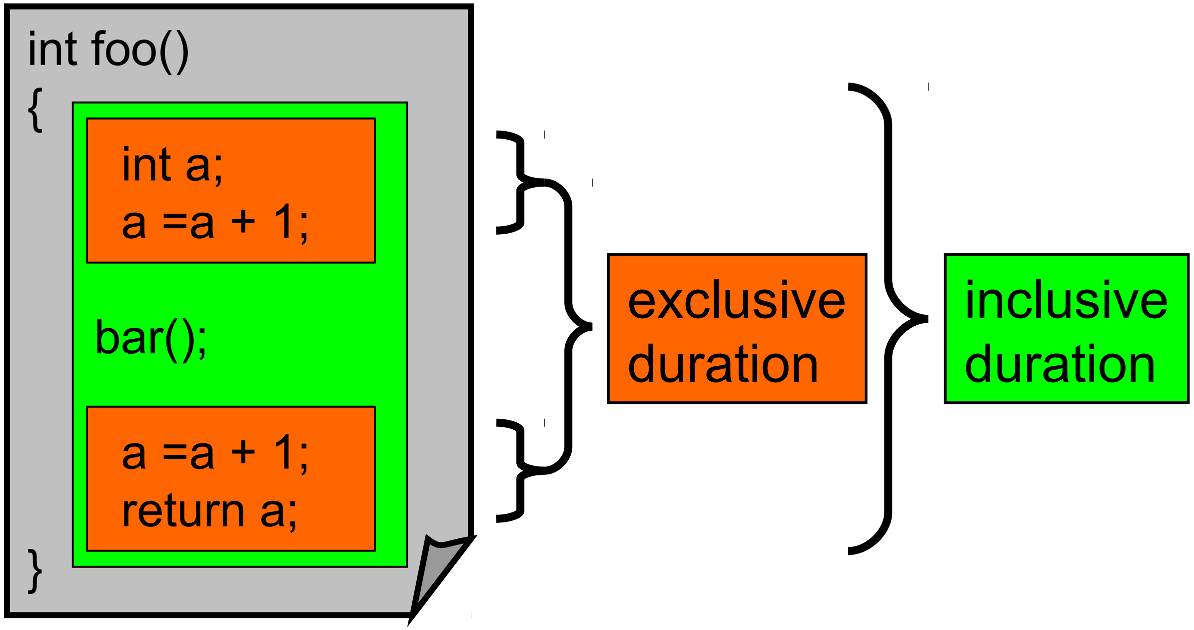
- Supports both direct and indirect performance observation
  - Direct instrumentation of program (system) code (probes)
  - Instrumentation invokes performance measurement
  - Event measurement: performance data, meta-data, context
  - Indirect mode supports sampling based on periodic timer or hardware performance counter overflow based interrupts
- Support for user-defined events
  - ***Interval*** (Start/Stop) events to measure exclusive & inclusive duration
  - ***Atomic events*** (Trigger at a single point with data, e.g., heap memory)
    - Measures total, samples, min/max/mean/std. deviation statistics
  - ***Context events*** (are atomic events with executing context)
    - Measures above statistics for a given calling path

Direct Observation: Events

- Event types
  - Interval events (begin/end events)
    - Measures exclusive & inclusive durations between events
    - Metrics monotonically increase
  - Atomic events (trigger with data value)
    - Used to capture performance data state
    - Shows extent of variation of triggered values (min/max/mean)
- Code events
  - Routines, classes, templates
  - Statement-level blocks, loops

Inclusive and Exclusive Profiles

- Performance with respect to code regions
- Exclusive measurements for region only
- Inclusive measurements includes child regions



Interval Events, Atomic Events in TAU

```
NODE 0;CONTEXT 0;THREAD 0:
---------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs  Inclusive Name
              msec   total msec                          usec/call
---------------------------------------------------------------------------------------
100.0        0.187        1,105           1          44    1105659 int main(int, char **) C
 93.2        1,030        1,030           1           0    1030654 MPI_Init()
  5.9        0.879           65          40         320       1637 void func(int, int) C
  4.6           51           51          40           0       1277 MPI_Barrier()
  1.2           13           13         120           0        111 MPI_Recv()
  0.8            9            9           1           0       9328 MPI_Finalize()
  0.0        0.137        0.137         120           0          1 MPI_Send()
  0.0        0.086        0.086          40           0          2 MPI_Bcast()
  0.0        0.002        0.002           1           0          2 MPI_Comm_size()
  0.0        0.001        0.001           1           0          1 MPI_Comm_rank()
---------------------------------------------------------------------------------------

USER EVENTS Profile :NODE 0, CONTEXT 0, THREAD 0
---------------------------------------------------------------------------------------
NumSamples   MaxValue   MinValue  MeanValue  Std. Dev.  Event Name
---------------------------------------------------------------------------------------
       365  5.138E+04      44.39   3.09E+04  1.234E+04  Heap Memory Used (KB) : Entry
       365  5.138E+04       2064  3.115E+04   1.21E+04  Heap Memory Used (KB) : Exit
        40         40         40         40          0  Message size for broadcast
---------------------------------------------------------------------------------------
                                                                  27,1          1%
```

Interval events e.g., routines (start/stop) show ***duration***

Atomic events (triggered with value) show ***extent of variation***

(min/max/mean)

% export TAU_CALLPATH_DEPTH=0
% export TAU_TRACK_HEAP=1

Atomic Events, Context Events

```
---------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs  Inclusive Name
              msec   total msec                          usec/call
---------------------------------------------------------------------------------------
100.0        0.253        1,106           1          44    1106701 int main(int, char **) C
 93.2        1,031        1,031           1           0    1031311 MPI_Init()
  6.0            1           66          40         320       1650 void func(int, int) C
  5.7           63           63          40           0       1588 MPI_Barrier()
  0.8            9            9           1           0       9119 MPI_Finalize()
  0.1            1            1         120           0         10 MPI_Recv()
  0.0        0.141        0.141         120           0          1 MPI_Send()
  0.0        0.085        0.085          40           0          2 MPI_Bcast()
  0.0        0.001        0.001           1           0          1 MPI_Comm_size()
  0.0            0            0           1           0          0 MPI_Comm_rank()
---------------------------------------------------------------------------------------

USER EVENTS Profile :NODE 0, CONTEXT 0, THREAD 0
---------------------------------------------------------------------------------------
NumSamples   MaxValue   MinValue  MeanValue  Std. Dev.  Event Name
---------------------------------------------------------------------------------------
        40         40         40         40          0  Message size for broadcast
       365  5.139E+04      44.39  3.091E+04  1.234E+04  Heap Memory Used (KB) : Entry
        40  5.139E+04       3097  3.114E+04  1.227E+04  Heap Memory Used (KB) : Entry : MPI_Barrier()
        40  5.139E+04   1.13E+04  3.134E+04  1.187E+04  Heap Memory Used (KB) : Entry : MPI_Bcast()
         1       2067       2067       2067          0  Heap Memory Used (KB) : Entry : MPI_Comm_rank()
         1       2066       2066       2066          0  Heap Memory Used (KB) : Entry : MPI_Comm_size()
         1  5.139E+04  5.139E+04  5.139E+04  0.0006905  Heap Memory Used (KB) : Entry : MPI_Finalize()
         1      57.56      57.56      57.56          0  Heap Memory Used (KB) : Entry : MPI_Init()
       120  5.139E+04   1.13E+04  3.134E+04  1.187E+04  Heap Memory Used (KB) : Entry : MPI_Recv()
       120  5.139E+04  1.129E+04  3.134E+04  1.187E+04  Heap Memory Used (KB) : Entry : MPI_Send()
         1      44.39      44.39      44.39          0  Heap Memory Used (KB) : Entry : int main(int, char **) C
        40  5.036E+04       2068  3.011E+04  1.227E+04  Heap Memory Used (KB) : Entry : void func(int, int) C
                                                                                 4,9          1%
```

Atomic events

Context events =atomic event + executing context

% export TAU_CALLPATH_DEPTH=1

Controls depth of executing context shown in profiles

% export TAU_TRACK_HEAP=1

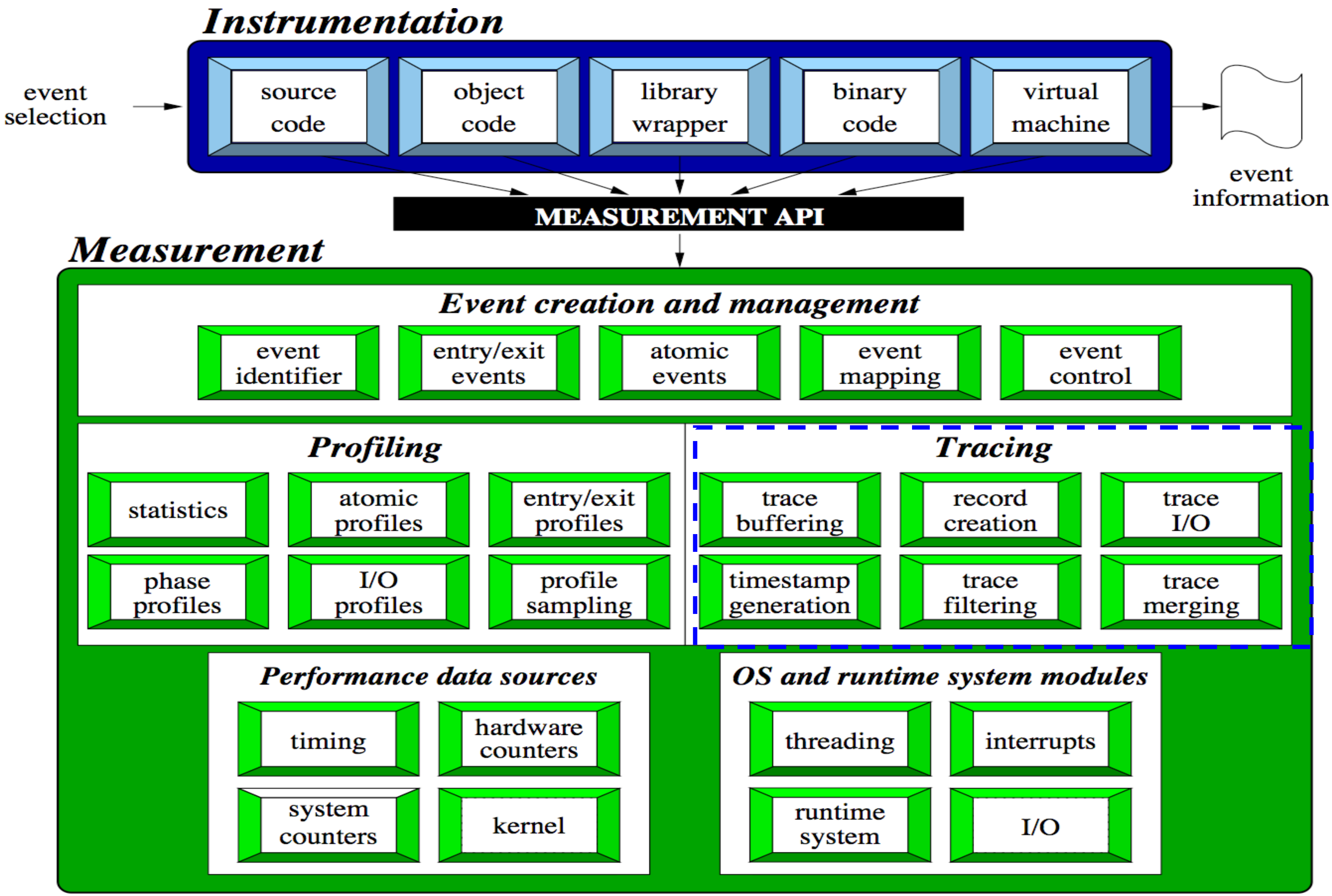
Context Events (Default)

```
xterm
NODE 0;CONTEXT 0;THREAD 0:
---------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs  Inclusive Name
              msec   total msec                          usec/call
---------------------------------------------------------------------------------------
100.0        0.357        1,114           1          44    1114040 int main(int, char **) C
 92.6        1,031        1,031           1           0    1031066 MPI_Init()
  6.7           72           74          40         320       1865 void func(int, int) C
  0.7            8            8           1           0       8002 MPI_Finalize()
  0.1            1            1         120           0         12 MPI_Recv()
  0.1        0.608        0.608          40           0         15 MPI_Barrier()
  0.0        0.136        0.136         120           0          1 MPI_Send()
  0.0        0.095        0.095          40           0          2 MPI_Bcast()
  0.0        0.001        0.001           1           0          1 MPI_Comm_size()
  0.0            0            0           1           0          0 MPI_Comm_rank()
---------------------------------------------------------------------------------------

USER EVENTS Profile :NODE 0, CONTEXT 0, THREAD 0
---------------------------------------------------------------------------------------
NumSamples   MaxValue   MinValue  MeanValue  Std. Dev.  Event Name
---------------------------------------------------------------------------------------
       365  5.139E+04      44.39  3.091E+04  1.234E+04  Heap Memory Used (KB) : Entry
         1      44.39      44.39      44.39          0  Heap Memory Used (KB) : Entry : int main(int, char **) C
         1       2068       2068       2068          0  Heap Memory Used (KB) : Entry : int main(int, char **) C => MPI_Comm_rank()
         1       2066       2066       2066          0  Heap Memory Used (KB) : Entry : int main(int, char **) C => MPI_Comm_size()
         1  5.139E+04  5.139E+04  5.139E+04          0  Heap Memory Used (KB) : Entry : int main(int, char **) C => MPI_Finalize()
         1      57.58      57.58      57.58          0  Heap Memory Used (KB) : Entry : int main(int, char **) C => MPI_Init()
        40  5.036E+04       2069  3.011E+04  1.228E+04  Heap Memory Used (KB) : Entry : int main(int, char **) C => void func(int, int) C
        40  5.139E+04       3098  3.114E+04  1.227E+04  Heap Memory Used (KB) : Entry : void func(int, int) C => MPI_Barrier()
        40  5.139E+04    1.13E+04  3.134E+04  1.187E+04  Heap Memory Used (KB) : Entry : void func(int, int) C => MPI_Bcast()
       120  5.139E+04    1.13E+04  3.134E+04  1.187E+04  Heap Memory Used (KB) : Entry : void func(int, int) C => MPI_Recv()
       120  5.139E+04    1.13E+04  3.134E+04  1.187E+04  Heap Memory Used (KB) : Entry : void func(int, int) C => MPI_Send()
       365  5.139E+04       2065  3.116E+04   1.21E+04  Heap Memory Used (KB) : Exit
```

```
% export TAU_CALLPATH_DEPTH=2
% export TAU_TRACK_HEAP=1
```

Context event =atomic event + executing context

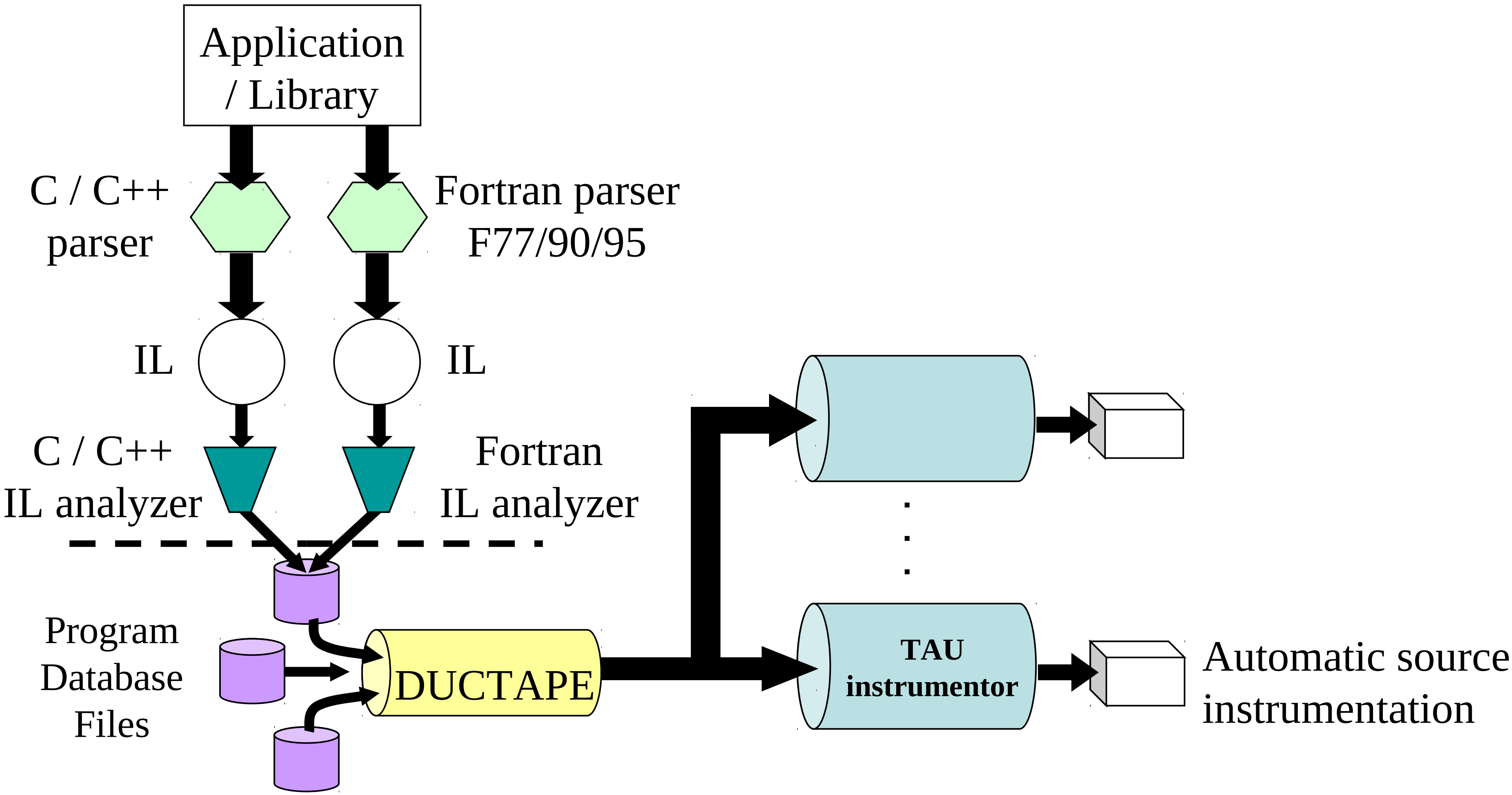
TAU Instrumentation / Measurement



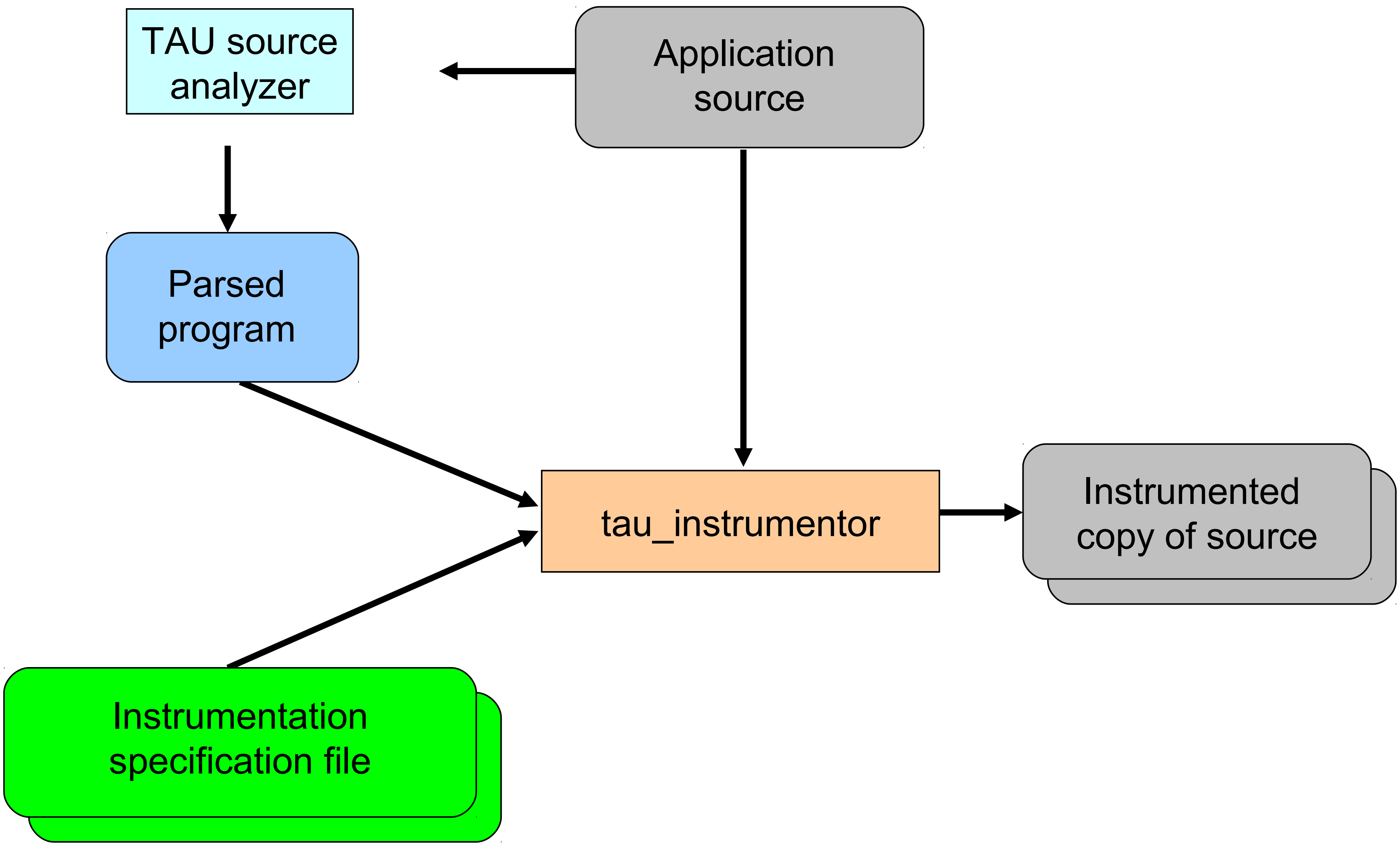
Direct Instrumentation Options in TAU

- Source Code Instrumentation
  - Manual instrumentation
  - Automatic instrumentation using pre-processor based on static analysis of source code (PDT), creating an instrumented copy
  - Compiler generates instrumented object code
- Library Level Instrumentation
  - Wrapper libraries for standard MPI libraries using PMPI interface
  - Wrapping external libraries where source is not available
- Runtime pre-loading and interception of library calls
- Binary Code instrumentation
  - Rewrite the binary, runtime instrumentation
- Virtual Machine, Interpreter, OS level instrumentation

TAU's Static Analysis System: Program Database Toolkit (PDT)



Automatic Source Instrumentation using PDT



PDT: Automatic Source Code Instrumentation

- To instrument source code using PDT
  - Choose an appropriate TAU stub makefile from <taudir>/<arch>/lib/Makefile.tau*:
    (typically, *arch*=i386_linux, x86_64, craycnl, bgp, cygwin … and *taudir*=/usr/local/packages/tau on LiveDVD)

  % export TAU_MAKEFILE=$TAU/Makefile.tau-mpi-pdt

  % make CC=tau_cc.sh CXX=tau_cxx.sh F90=tau_f90.sh

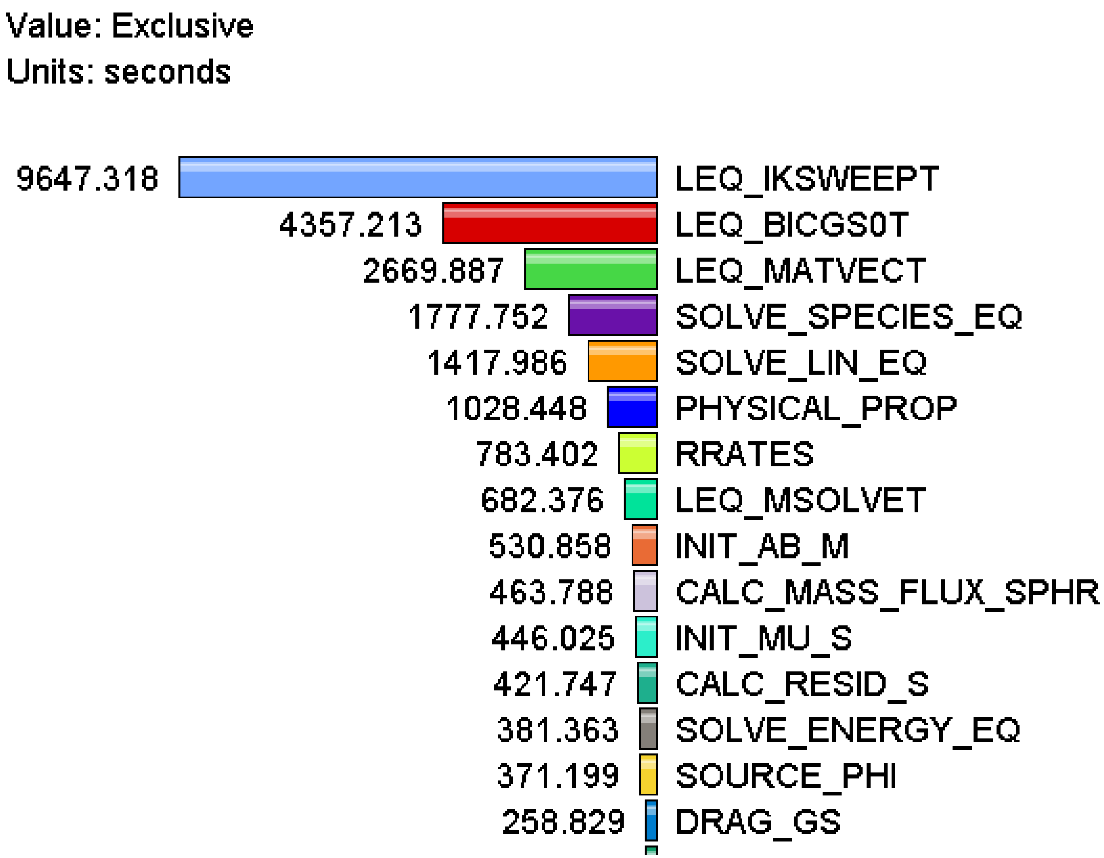
- Execute application and analyze performance data:

  % pprof (for text based profile display)

  % paraprof (for GUI)

Usage Scenarios: Routine Level Profile

- How much time is spent in each application routine?



Solution: Generating a flat profile with MPI

```
% export TAU_MAKEFILE=<taudir>/<arch>/lib/Makefile.tau-mpi-pdt
% export PATH=<taudir>/<arch>/bin:$PATH
Or
% module load tau
% make F90=tau_f90.sh
Or
% tau_f90.sh matmult.f90
% mpirun -np 8 ./a.out
% paraprof
To view. To view the data locally on the workstation,
% paraprof --pack app.ppk
  Move the app.ppk file to your desktop.

% paraprof app.ppk

Click on the "node 0" label to see profile for that node. Right
click to see other options. Windows -> 3D Visualization for 3D
window.
```

Automatic Instrumentation

- We now provide compiler wrapper scripts
  - Simply replace `CC` with `tau_cxx.sh`
  - Automatically instruments C++ and C source code, links with TAU MPI Wrapper libraries.
- Use `tau_cc.sh` and `tau_f90.sh` for C and Fortran

Before

```
CXX = mpicxx
F90 = mpif90
CXXFLAGS =
LIBS = -lm
OBJS = f1.o f2.o f3.o … fn.o

app: $(OBJS)
     $(CXX) $(LDFLAGS) $(OBJS) -o $@
     $(LIBS)
.cpp.o:
     $(CXX) $(CXXFLAGS) -c $<
```

After

```
CXX = tau_cxx.sh
F90 = tau_f90.sh
CXXFLAGS =
LIBS = -lm
OBJS = f1.o f2.o f3.o … fn.o

app: $(OBJS)
     $(CXX) $(LDFLAGS) $(OBJS) -o $@
     $(LIBS)
.cpp.o:
     $(CXX) $(CXXFLAGS) -c $<
```

Passing Optional Parameters to TAU Compiler Scripts

- See `<taudir>/<arch>/bin/tau_compiler.sh -help`
- Compilation:

```
% ftn -c foo.f90
```

Changes to

```
% gfparse foo.f90 $(OPT1)
% tau_instrumentor foo.pdb foo.f90 –o foo.inst.f90 $(OPT2)
% ftn –c foo.inst.f90 –o foo.o $(OPT3)
```

- Linking:

```
% ftn foo.o bar.o –o app
```

Changes to

```
% ftn foo.o bar.o –o app <taulibs> $(OPT4)
```

- Where options OPT[1-4] default values may be overridden by the user:

```
F90 = tau_f90.sh
```

Compile-Time Environment Variables

- Optional parameters for the TAU_OPTIONS environment variable:
  % tau_compiler.sh

-optVerbose	Turn on verbose debugging messages
-optCompInst	Use compiler based instrumentation
-optMemDbg	Enable memory debugging instrumentation.
-optTrackIO	Wrap POSIX I/O call and calculates vol/bw of I/O operations (Requires TAU to be configured with –*iowrapper*)
-optKeepFiles	Does not remove intermediate .pdb and .inst.* files
-optPreProcess	Preprocess Fortran sources before instrumentation
-optTauSelectFile=”<file>"	Specify selective instrumentation file for *tau_instrumentor*
-optTauWrapFile=”<file>"	Specify path to *link_options.tau* generated by *tau_gen_wrapper*
-optHeaderInst	Enable Instrumentation of headers
-optLinking=""	Options passed to the linker. Typically $(TAU_MPI_FLIBS) $(TAU_LIBS) $(TAU_CXXLIBS)
-optCompile=""	Options passed to the compiler. Typically $(TAU_MPI_INCLUDE) $(TAU_INCLUDE) $(TAU_DEFS)
-optPdtF95Opts=""	Add options for Fortran parser in PDT (f95parse/gfparse)
-optPdtF95Reset=""	Reset options for Fortran parser in PDT (f95parse/gfparse)
-optPdtCOpts=""	Options for C parser in PDT (cparse). Typically $(TAU_MPI_INCLUDE) $(TAU_INCLUDE) $(TAU_DEFS)
-optPdtCxxOpts=""	Options for C++ parser in PDT (cxxparse). Typically $(TAU_MPI_INCLUDE) $(TAU_INCLUDE) $(TAU_DEFS) ...

Compiling Fortran Codes with TAU

- If your Fortran code uses free format in .f files (fixed is default for .f), you may use:

```
% export TAU_OPTIONS='-optPdtF95Opts="-R free" -optVerbose '
```

- To use the compiler based instrumentation instead of PDT (source-based):

```
% export TAU_OPTIONS='-optCompInst -optVerbose'
```

- If your Fortran code uses C preprocessor directives (#include, #ifdef, #endif):

```
% export TAU_OPTIONS='-optPreProcess -optVerbose -optDetectMemoryLeaks'
```

- To use an instrumentation specification file:

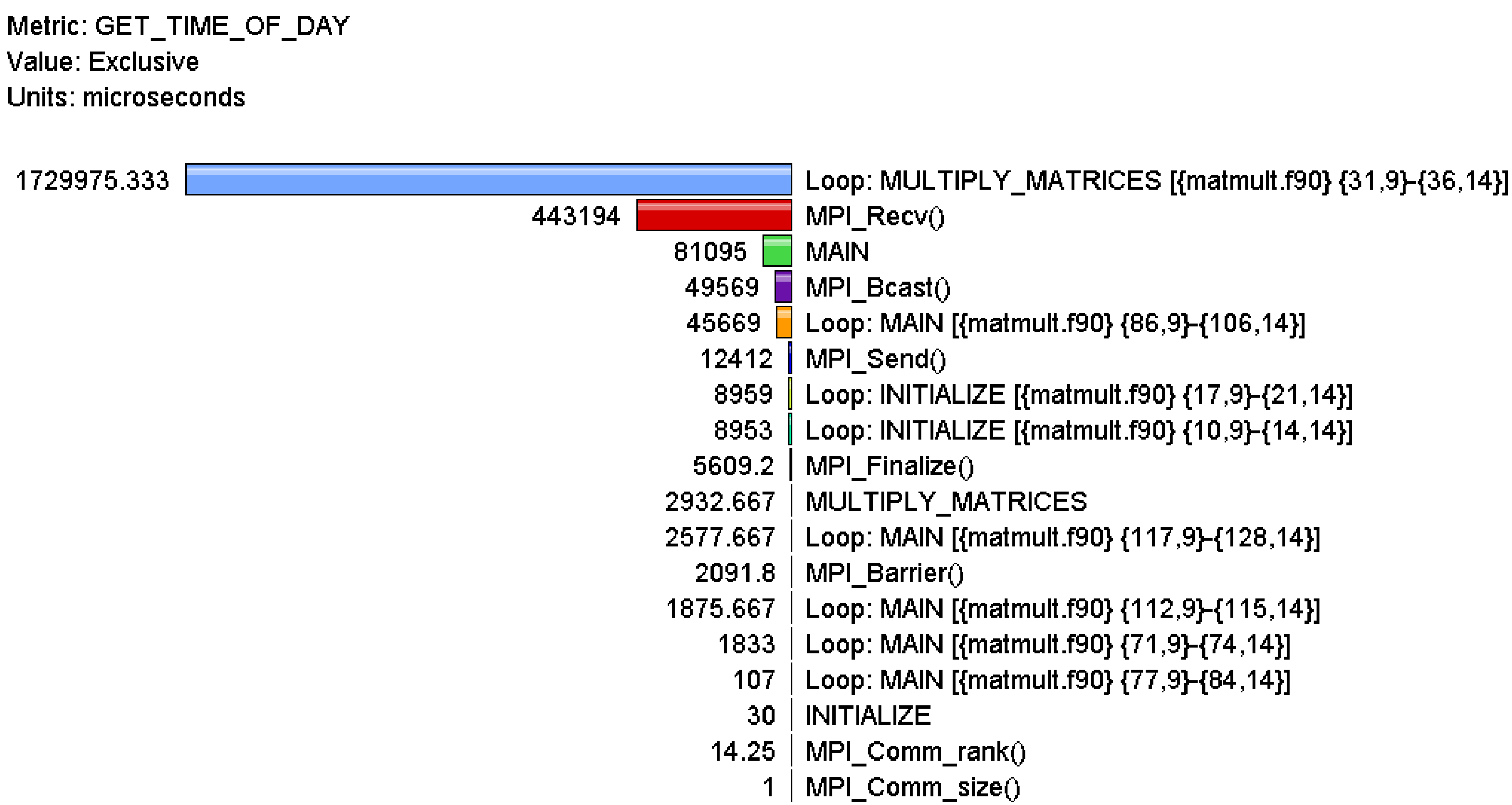
```
% export TAU_OPTIONS='-optTauSelectFile=select.tau -optVerbose -optPreProcess'
% cat select.tau
BEGIN_INSTRUMENT_SECTION
loops  routine="#"
# this statement instruments all outer loops in all routines. # is wildcard as well as comment in first column.
END_INSTRUMENT_SECTION
```

Runtime Environment Variables in TAU

Environment Variable	Default	Description
TAU_TRACE	0	Setting to 1 turns on tracing
TAU_CALLPATH	0	Setting to 1 turns on callpath profiling
TAU_TRACK_MEMORY_LEAKS	0	Setting to 1 turns on leak detection (for use with tau_exec –memory ./a.out)
TAU_TRACK_HEAP or TAU_TRACK_HEADROOM	0	Setting to 1 turns on tracking heap memory/headroom at routine entry & exit using context events (e.g., Heap at Entry: main=>foo=>bar)
TAU_CALLPATH_DEPTH	2	Specifies depth of callpath. Setting to 0 generates no callpath or routine information, setting to 1 generates flat profile and context events have just parent information (e.g., Heap Entry: foo)
TAU_TRACK_IO_PARAMS	0	Setting to 1 with –optTrackIO or tau_exec –io captures arguments of I/O calls
TAU_SAMPLING	1	Generates sample based profiles
TAU_COMM_MATRIX	0	Setting to 1 generates communication matrix display using context events
TAU_THROTTLE	1	Setting to 0 turns off throttling. Enabled by default to remove instrumentation in lightweight routines that are called frequently
TAU_THROTTLE_NUMCALLS	100000	Specifies the number of calls before testing for throttling
TAU_THROTTLE_PERCALL	10	Specifies value in microseconds. Throttle a routine if it is called over 100000 times and takes less than 10 usec of inclusive time per call
TAU_COMPENSATE	0	Setting to 1 enables runtime compensation of instrumentation overhead
TAU_PROFILE_FORMAT	Profile	Setting to “merged” generates a single file. “snapshot” generates xml format

Usage Scenarios: Loop Level Instrumentation

- Goal: What loops account for the most time? How much?
- Flat profile with wallclock time with loop instrumentation:

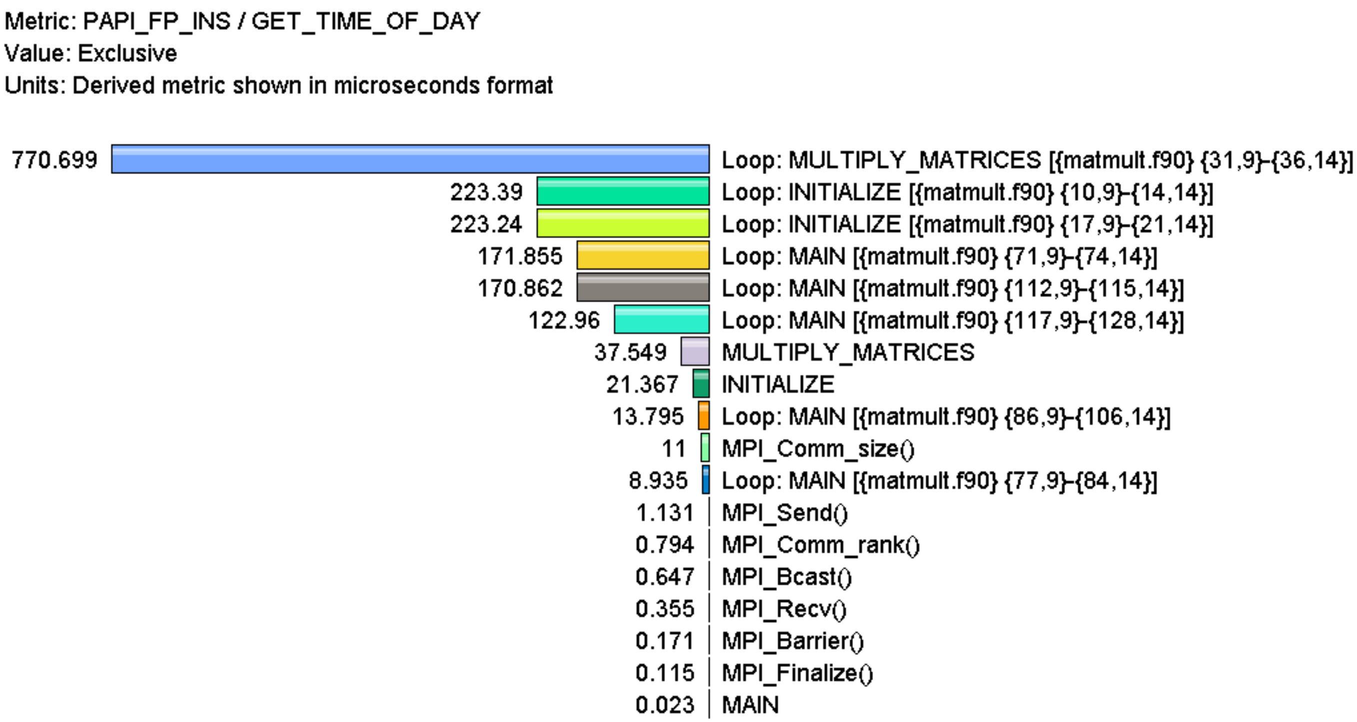


Solution: Generating a loop level profile

```
% export TAU_MAKEFILE=<taudir>/<arch>/lib/Makefile.tau-mpi-pdt
% export TAU_OPTIONS='-optTauSelectFile=select.tau -optVerbose'
% cat select.tau
  BEGIN_INSTRUMENT_SECTION
  loops routine="#"
  END_INSTRUMENT_SECTION

% module load tau
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)
% mpirun -np 8 ./a.out
% paraprof --pack app.ppk
  Move the app.ppk file to your desktop.

% paraprof app.ppk
```

Computing Floating Point Instructions Executed Per Second in Loops

- Goal: What execution rate do my application loops get in mflops?
- Flat profile with PAPI_FP_INS and time with loop instrumentation:

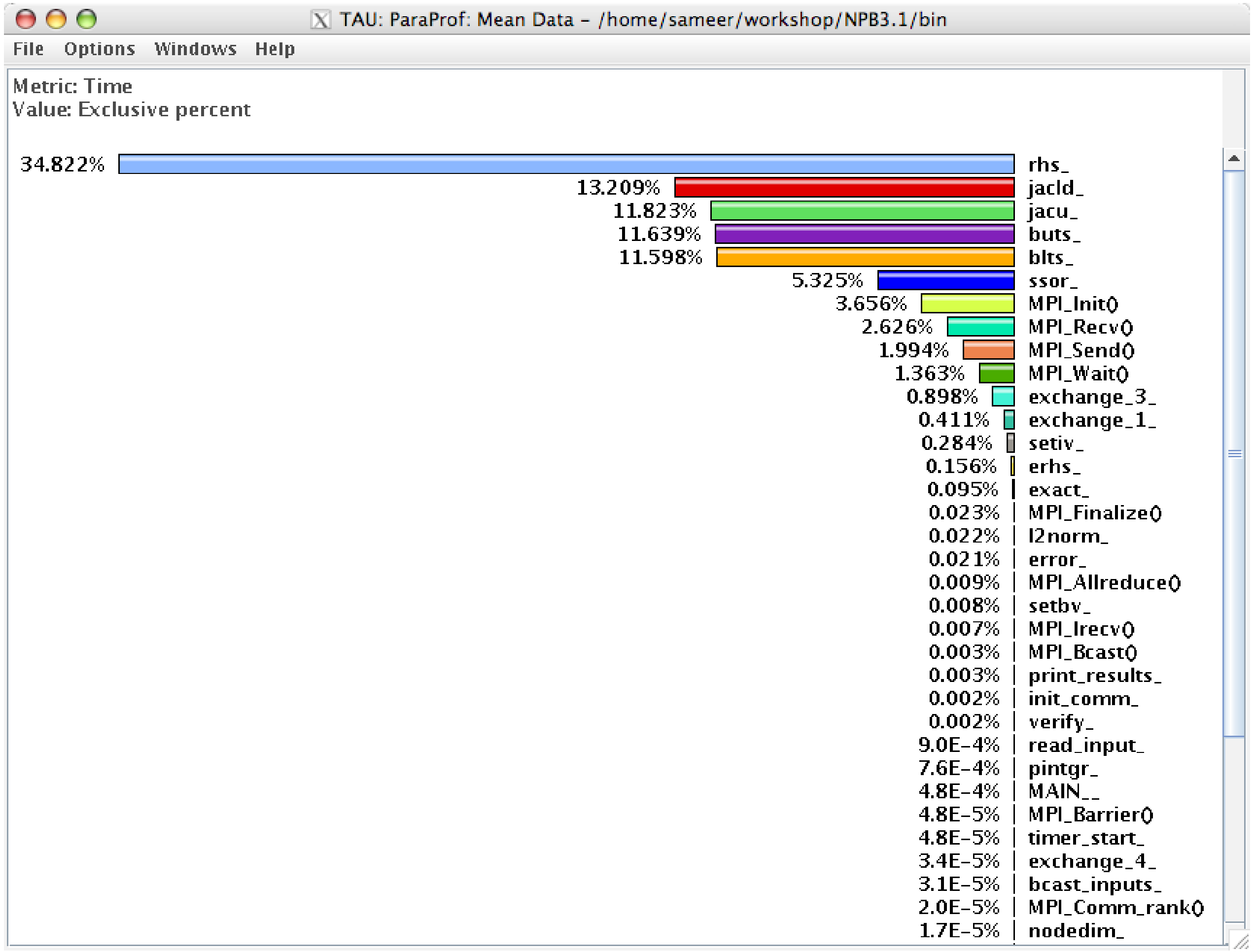


Generate a PAPI profile with 2 or more counters

```
% export TAU_MAKEFILE=<taudir>/<arch>/lib/Makefile.tau-papi-mpi-pdt
% export TAU_OPTIONS='-optTauSelectFile=select.tau -optVerbose'
% cat select.tau
  BEGIN_INSTRUMENT_SECTION
  loops routine="#"
  END_INSTRUMENT_SECTION

% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)
% export TAU_METRICS=TIME:PAPI_FP_INS
% mpirun -np 8 ./a.out
% paraprof --pack app.ppk
  Move the app.ppk file to your desktop.
% paraprof app.ppk
  Choose Options -> Show Derived Panel -> Click PAPI_FP_INS,
 Click "/",  Click TIME, Apply, Choose new metric by double clicking.
```

Usage Scenarios: Compiler-based Instrumentation

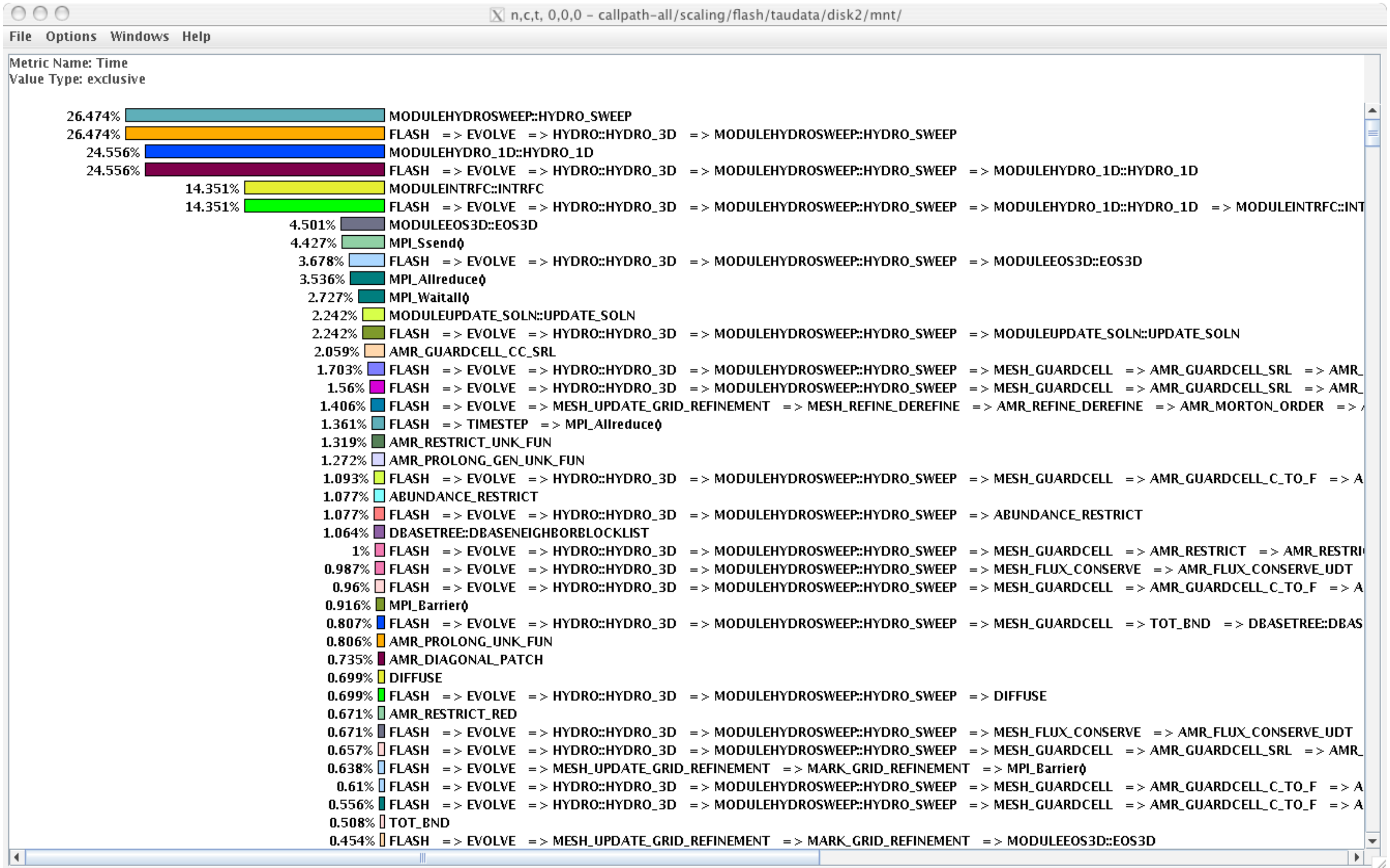
- Use the compiler to automatically emit instrumentation calls in the object code instead of parsing the source code using PDT.



Use Compiler-Based Instrumentation

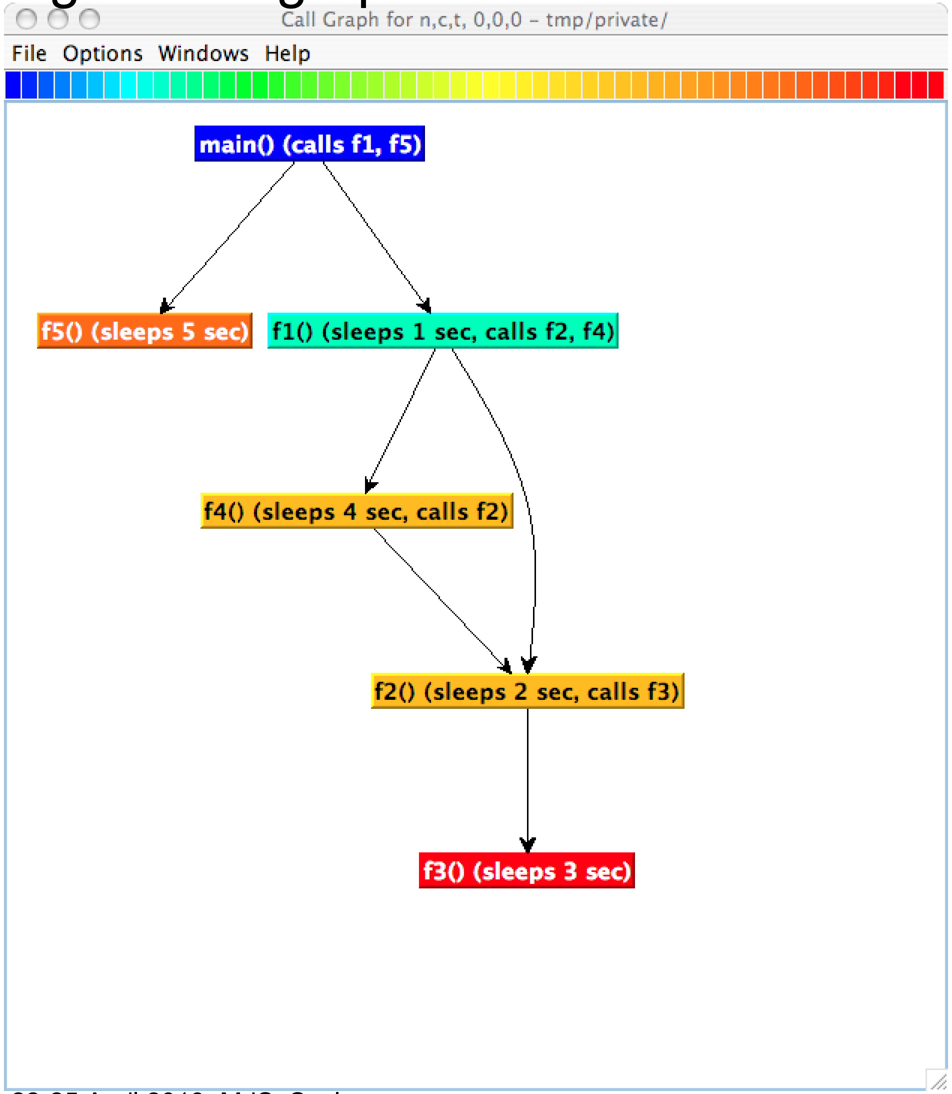
```
% export TAU_MAKEFILE=<taudir>/<arch>/lib/Makefile.tau-mpi-pdt
% export TAU_OPTIONS='-optCompInst -optQuiet'

% make CC=tau_cc.sh CXX=tau_cxx.sh F90=tau_f90.sh

NOTE: You may also use the short-hand scripts taucc, tauf90,
taucxx instead of specifying TAU_OPTIONS and using the traditional
tau_<cc,cxx,f90>.sh scripts. These scripts use compiler-based
instrumentation by default.

% make CC=taucc CXX=taucxx F90=tauf90
% mpirun -np 8 ./a.out
% paraprof --pack app.ppk
  Move the app.ppk file to your desktop.
% paraprof app.ppk
```

Generate a Callpath Profile



Callpath Profile

- Generates program callgraph

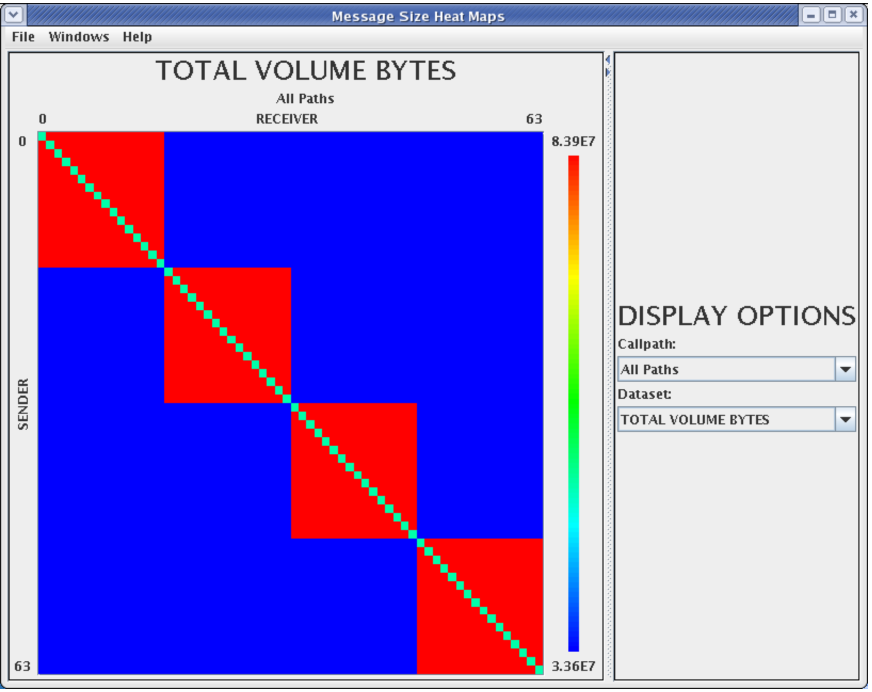


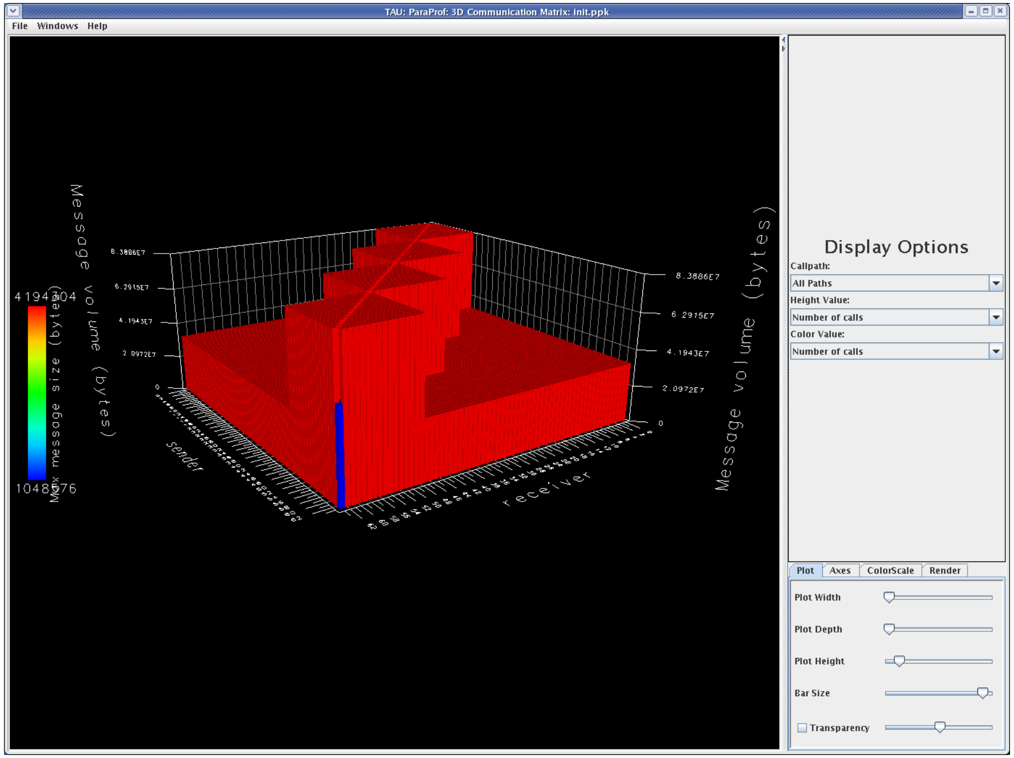
Generate a Callpath Profile

```
% export TAU_MAKEFILE=<taudir>/<arch>/lib/Makefile.tau-mpi-pdt
% export PATH=<taudir>/<arch>/bin:$PATH
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)

% export TAU_CALLPATH=1
% export TAU_CALLPATH_DEPTH=100
(truncates all calling paths to a specified depth)


% mpirun -np 8 ./a.out
% paraprof --pack app.ppk
  Move the app.ppk file to your desktop.
% paraprof app.ppk
(Windows -> Thread -> Call Graph)
```

Communication Matrix Display

- Goal: What is the volume of inter-process communication? Along which calling path?





Evaluate Scalability using PerfExplorer Charts

```
% export TAU_MAKEFILE=$TAU/Makefile.tau-mpi-pdt
% export PATH=<taudir>/<arch>/bin:$PATH
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)
% export TAU_COMM_MATRIX=1

% mpirun -np 8 ./a.out

% paraprof
(Windows -> Communication Matrix)
(Windows -> 3D Communication Matrix)
```

Three Instrumentation Techniques for Wrapping External Libraries

- Pre-processor based substitution by re-defining a call (e.g., read)
  - Tool defined header file with same name <*unistd.h*> takes precedence
  - Header redefines a routine as a different routine using macros
  - Substitution: *read()* substituted by preprocessor as *tau_read()* at callsite
- Preloading a library at runtime
  - Library preloaded (*LD_PRELOAD* env var in Linux) in the address space of executing application intercepts calls from a given library
  - Tool's wrapper library defines *read(),* gets address of global *read()* symbol (dlsym), internally calls timing calls around call to global read
- Linker based substitution
  - Wrapper library defines __*wrap_read* which calls __*real_read* and linker is passed *-Wl,-wrap,read* to substitute all references to read from application's object code with the __*wrap_read* defined by the tool

Issues: Preprocessor based substitution

- Pre-processor based substitution by re-defining a call
  - Compiler replaces read() with tau_read() in the body of the source code
- Advantages:
  - Simple to instrument
    - Preprocessor based replacement
    - A header file redefines the calls
    - No special linker or runtime flags required
- Disadvantages
  - Only works for C & C++ for replacing calls in the body of the code.
  - Incomplete instrumentation: fails to capture calls in uninstrumented libraries (e.g., libhdf5.a)

Issues: Linker based substitution

- Linker based substitution
  - Wrapper library defines __wrap_read which calls __real_read and linker is passed -Wl,-wrap, read
- Advantages
  - Tool can intercept all references to a given call
  - Works with static as well as dynamic executables
  - No need to recompile the application source code, just re-link the application objects and libraries with the tool wrapper library
- Disadvantages
  - Wrapping an entire library can lengthen the linker command line with multiple –Wl,-wrap,<func> arguments. It is better to store these arguments in a file and pass the file to the linker
  - Approach does not work with un-instrumented binaries

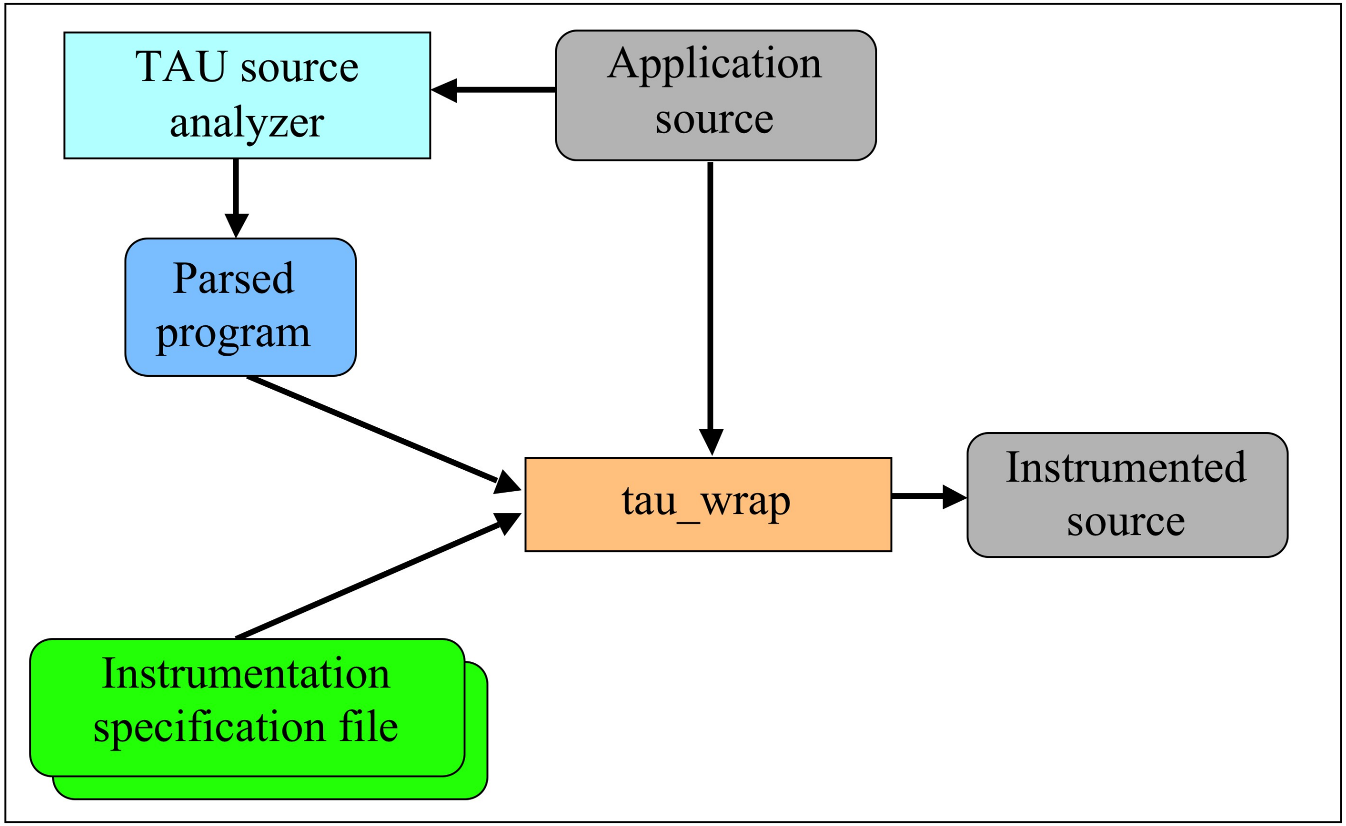
Solution: tau_gen_wrapper

- Automates creation of wrapper libraries using TAU
- Input:
  - header file (foo.h)
  - library to be wrapped (/path/to/libfoo.a)
  - technique for wrapping
    - Preprocessor based redefinition (-d)
    - Runtime preloading (-r)
    - Linker based substitution (-w: default)
  - Optional selective instrumentation file (-f select)
    - Exclude list of routines, or
    - Include list of routines
- Output:
  - wrapper library
  - optional *link_options.tau* file (-w), pass –optTauWrapFile=<file> in TAU_OPTIONS environment variable

Design of wrapper generator (tau_gen_wrapper)

- *tau_gen_wrapper* shell script:
  - parses source of header file using static analysis tool Program Database Toolkit (PDT)
  - Invokes *tau_wrap*, a tool that generates
    - instrumented wrapper code,
    - an optional *link_options.tau* file (for linker-based substitution, -w)
    - Makefile for compiling the wrapper interposition library
  - Builds the wrapper library using make
- Use TAU_OPTIONS environment variable to pass location of link_options.tau file using
  % export TAU_OPTIONS='-optTauWrapFile=<path/to/link_options.tau> -optVerbose'
- Use *tau_exec –loadlib=<wrapperlib.so>* to pass location of wrapper library for preloading based substitution

tau_wrap


TAU source analyzer
Application source
Parsed program
tau_wrap
Instrumented source
Instrumentation specification file

HDF5 Library Wrapping

```
[sameer@zorak]$ tau_gen_wrapper hdf5.h /usr/lib/libhdf5.a -f select.tau

Usage : tau_gen_wrapper <header> <library> [-r|-d|-w (default)] [-g groupname] [-i headerfile] [-c|-c++|-fortran] [-f <instr_spec_file> ]
• instruments using runtime preloading (-r), or -Wl,-wrap linker (-w), redirection of header file to redefine the wrapped routine (-d)
• instrumentation specification file (select.tau)
• group (hdf5)
• tau_exec loads libhdf5_wrap.so shared library using –loadlib=<libwrap_pkg.so>
• creates the wrapper/ directory

NODE 0;CONTEXT 0;THREAD 0:
---------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs  Inclusive Name
              msec   total msec                          usec/call
---------------------------------------------------------------------------------------
100.0        0.057            1           1          13       1236 .TAU Application
 70.8        0.875        0.875           1           0        875 hid_t H5Fcreate()
  9.7         0.12         0.12           1           0        120 herr_t H5Fclose()
  6.0        0.074        0.074           1           0         74 hid_t H5Dcreate()
  3.1        0.038        0.038           1           0         38 herr_t H5Dwrite()
  2.6        0.032        0.032           1           0         32 herr_t H5Dclose()
  2.1        0.026        0.026           1           0         26 herr_t H5check_version()
  0.6        0.008        0.008           1           0          8 hid_t H5Screate_simple()
  0.2        0.002        0.002           1           0          2 herr_t H5Tset_order()
  0.2        0.002        0.002           1           0          2 hid_t H5Tcopy()
  0.1        0.001        0.001           1           0          1 herr_t H5Sclose()
  0.1        0.001        0.001           2           0          0 herr_t H5open()
  0.0            0            0           1           0          0 herr_t H5Tclose()
```

Using POSIX I/O wrapper library in TAU

- Setting environment variable TAU_OPTIONS=-optTrackIO links in TAU's wrapper interposition library using linker-based substitution
- Instrumented application generates bandwidth, volume data
- Workflow:
  - % export TAU_OPTIONS='-optTrackIO –optVerbose'
  - % export TAU_MAKEFILE=/path/to/tau/x86_64/lib/Makefile.tau-mpi-pdt
  - % make CC=tau_cc.sh CXX=tau_cxx.sh F90=tau_f90.sh
  - % mpirun –np 8 ./a.out
  - % paraprof
- Get additional data regarding individual arguments by setting environment variable TAU_TRACK_IO_PARAMS=1 prior to running

Issues: Preloading a wrapper library at runtime

- Preloading a library at runtime
  - Tool defines read(), gets address of global read() symbol (dlsym), internally calls timing calls around call to global read
  - *tau_exec* tool uses this mechanism to intercept library calls
- Advantages
  - No need to re-compile or re-link the application source code
  - Drop-in replacement library implemented using LD_PRELOAD environment variable under Linux, Cray CNL, IBM BG/P CNK, Solaris…
- Disadvantages
  - Only works with dynamic executables. Default compilation mode under Cray XE6 and IBM BG/P is to use static executables
  - Not all operating systems support preloading of dynamic shared objects (DSOs)

Runtime Preloading: tau_exec

- Runtime instrumentation by pre-loading the measurement library
- Works on dynamic executables (default under Linux)
- Can substitute I/O, MPI, SHMEM, CUDA, OpenCL, and memory allocation/deallocation routines with instrumented calls
- Track interval events (e.g., time spent in write()) as well as atomic events (e.g., how much memory was allocated) in wrappers
- Accurately measure I/O and memory usage
- Preload any wrapper interposition library in the context of the executing application

Preloading a Specific TAU Measurement Library

% ./configure –pdt=<dir> -mpi –papi=<dir>; make install
Creates in <taudir>/<arch>/lib:
Makefile.tau-papi-mpi-pdt
shared-papi-mpi-pdt/libTAU.so

% ./configure –pdt=<dir> -mpi; make install creates
Makefile.tau-mpi-pdt
shared-mpi-pdt/libTAU.so

To explicitly choose preloading of shared-<options>/libTAU.so change:
% mpirun –np 8 ./a.out to
% mpirun –np 8 tau_exec –T <comma_separated_options> ./a.out

% mpirun –np 8 tau_exec –T papi,mpi,pdt ./a.out
Preloads <taudir>/<arch>/shared-papi-mpi-pdt/libTAU.so
% mpirun –np 8 tau_exec –T papi ./a.out
Preloads <taudir>/<arch>/shared-papi-mpi-pdt/libTAU.so by matching.
% mpirun –np 8 tau_exec –T papi,mpi,pdt –s ./a.out
Does not execute the program. Just displays the library that it will preload if executed without the –s option.
NOTE: -mpi configuration is selected by default. Use –T serial for Sequential programs.

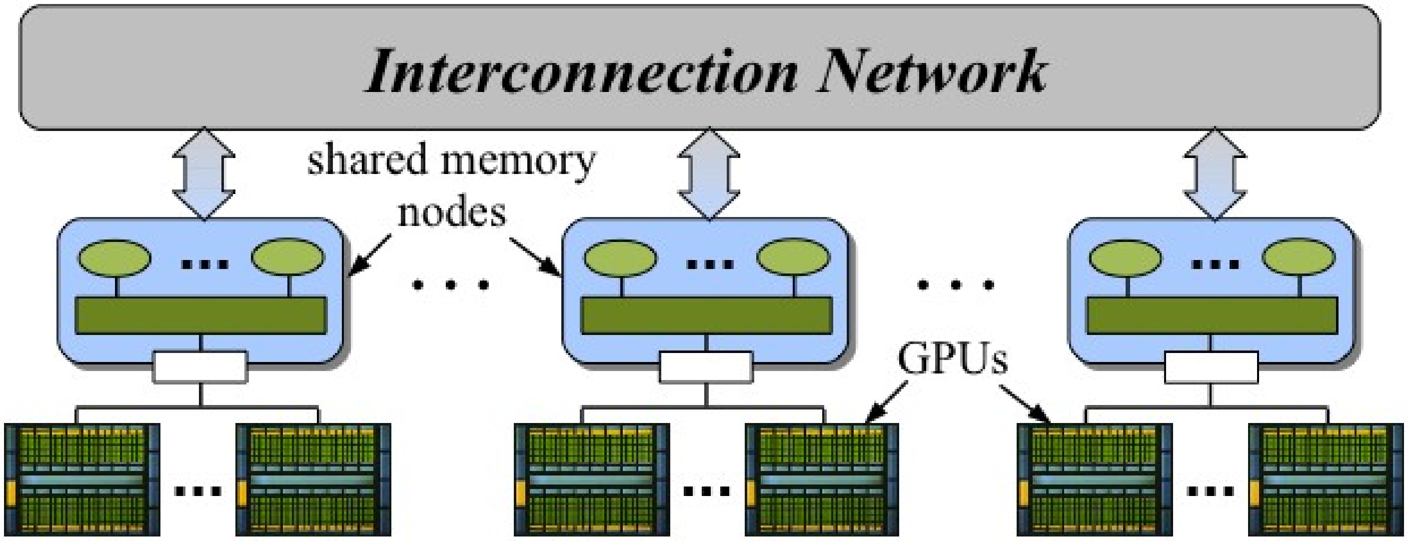
TAU Execution Command (tau_exec)

- Uninstrumented execution
  - % mpirun –np 8 ./a.out
- Track MPI performance
  - % mpirun –np 8 tau_exec ./a.out
- Track POSIX I/O and MPI performance (MPI enabled by default)
  - % mpirun –np 8 tau_exec –io ./a.out
- Track memory operations
  - % setenv TAU_TRACK_MEMORY_LEAKS 1
  - % mpirun –np 8 tau_exec –memory ./a.out
- Use event based sampling (compile with –g)
  - % mpirun –np 8 tau_exec –ebs ./a.out
  - Also –ebs_source=<PAPI_COUNTER> -ebs_period=<overflow_count>
- Load wrapper interposition library
  - % mpirun –np 8 tau_exec –loadlib=<path/libwrapper.so> ./a.out
- Track GPGPU operations
  - % mpirun –np 8 tau_exec –T serial –cupti ./a.out
  - % mpirun –np 8 tau_exec –opencl ./a.out

Profiling GPGPU Executions

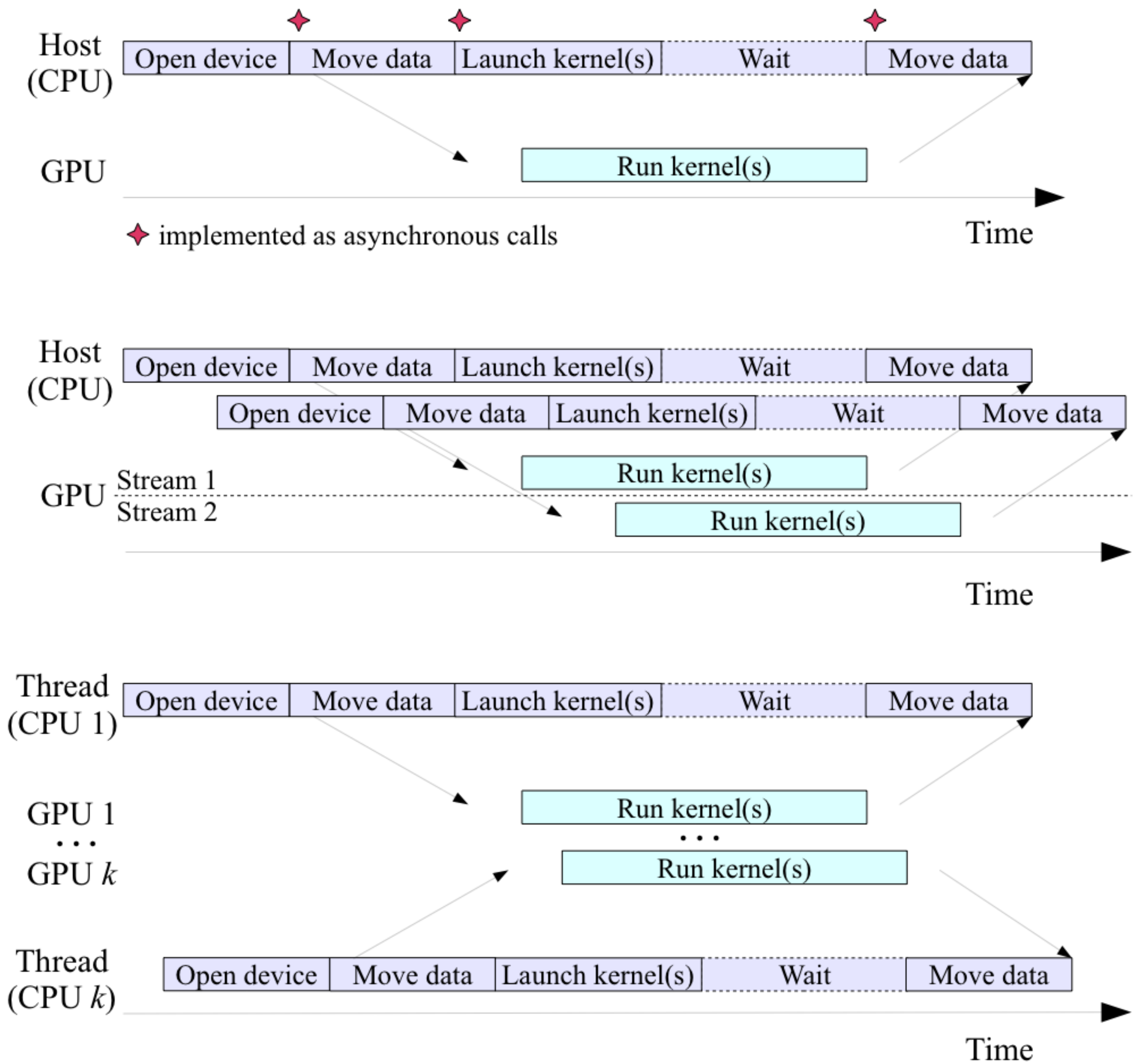
- GPGPU compilers (e.g., CAPS hmpp and PGI) can now automatically generate GPGPU code using manual annotation of loop-level constructs and routines (hmpp)
- The loops (and routines for HMPP) are transferred automatically to the GPGPU
- TAU intercepts the runtime library routines and examines the arguments
- Shows events as seen from the host
- Profiles and traces GPGPU execution

- Multi-CPU, multicore shared memory nodes
- GPU accelerators connected by high-BW I/O
- Cluster interconnection network



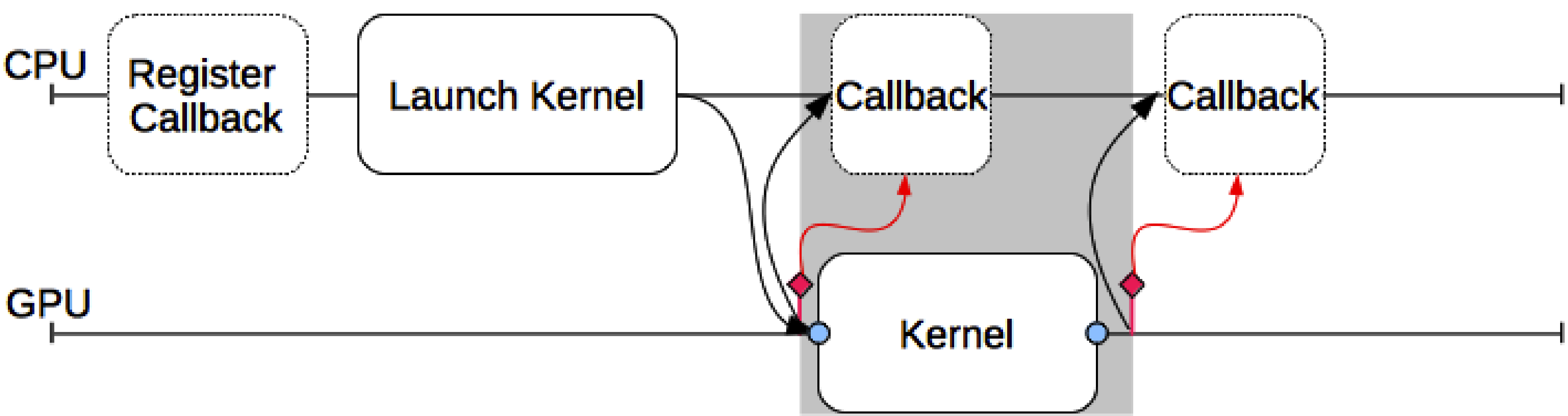
Host (CPU) - GPU Scenarios

- Single GPU
- Multi-stream
- Multi-CPU, Multi-GPU



Host-GPU Measurement – Callback Method

- GPU driver libraries provide callbacks for certain routines and captures measurements
- Measurement tool registers the callbacks and processes performance data
- Application code is not modified



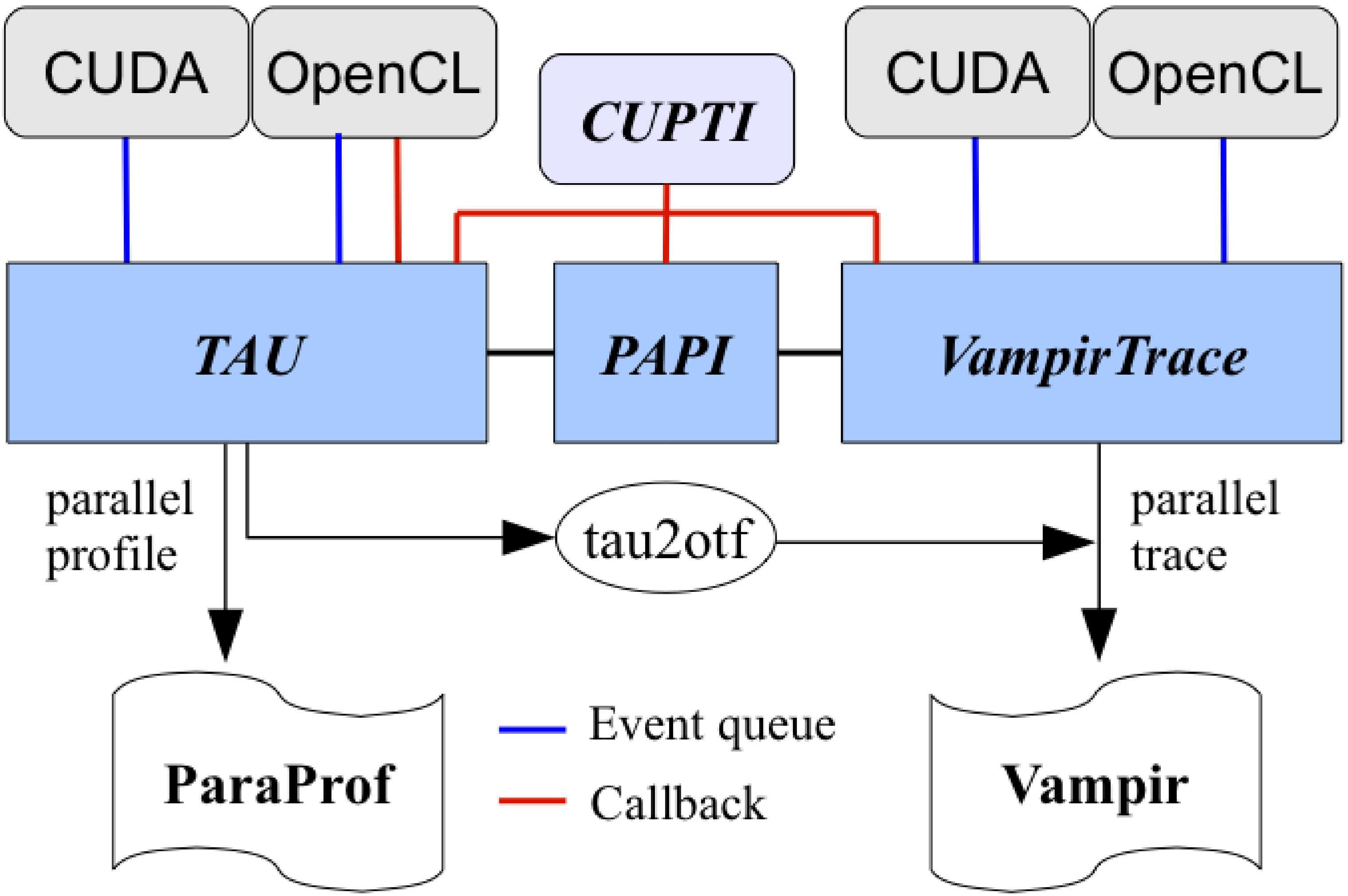
Method Support and Implementation

- Synchronous method
  - Place instrumentation appropriately around GPU calls (kernel launch, library routine, …)
  - Wrap (synchronous) library with performance tool
- Event queue method
  - Utilize CUDA and OpenCL event support
  - Again, need instrumentation to create and insert events in the streams with kernel launch and process events
  - Can be implemented with driver library wrapping
- Callback method
  - Utilize language-level callback support in OpenCL
  - Utilize NVIDIA CUDA Performance Tool Interface (CUPTI)
  - Need to appropriately register callbacks

GPU Performance Measurement Tools

- Support the Host-GPU performance perspective
- Provide integration with existing measurement system to facilitate tool use
- Utilize support in GPU driver library and device
- Tools
  - TAU performance system
  - Vampir
  - PAPI
  - NVIDIA CUPTI

GPU Performance Tool Interoperability



NVIDIA CUPTI

- NVIDIA is developing CUPTI to enable the creation of profiling and tracing tools
- Callback API
  - Interject tool code at the entry and exist to each CUDA runtime and driver API call
- Counter API
  - Query, configure, start, stop, and read the counters on CUDA-enabled devices
- CUPTI is delivered as a dynamic library
- CUPTI is released with CUDA 4.0

TAU for Heterogeneous Measurement

- Multiple performance perspectives
- Integrate Host-GPU support in TAU measurement framework
  - Enable use of each measurement approach
  - Include use of PAPI and CUPTI
  - Provide profiling and tracing support
- Tutorial
  - Use TAU library wrapping of libraries
  - Use `tau_exec` to work with binaries

    % ./a.out (uninstrumented)

    % tau_exec –T serial –cupti ./a.out

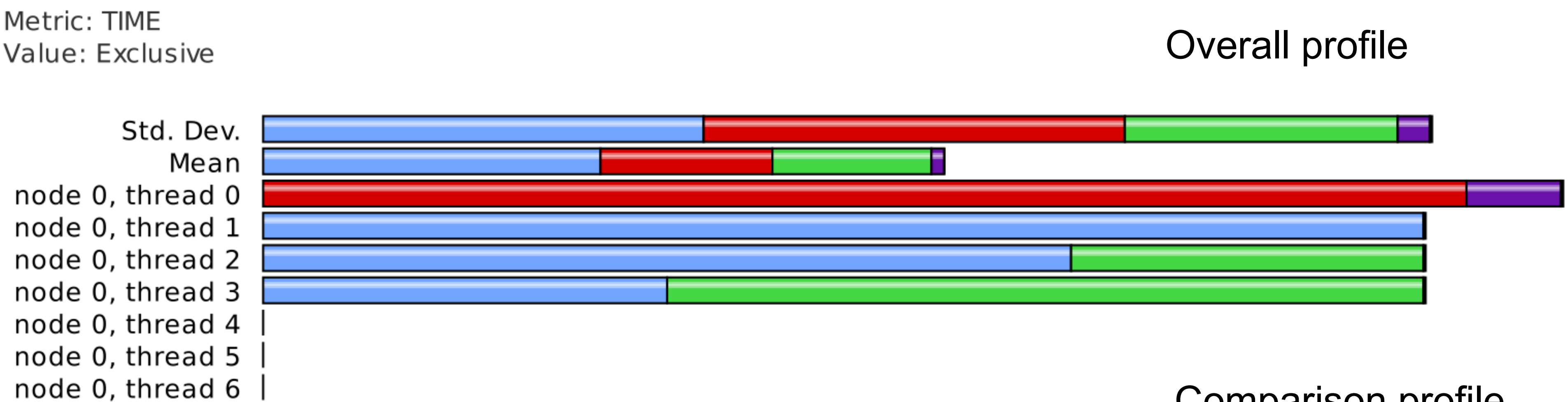
    % paraprof

Example: SDK simpleMultiGPU

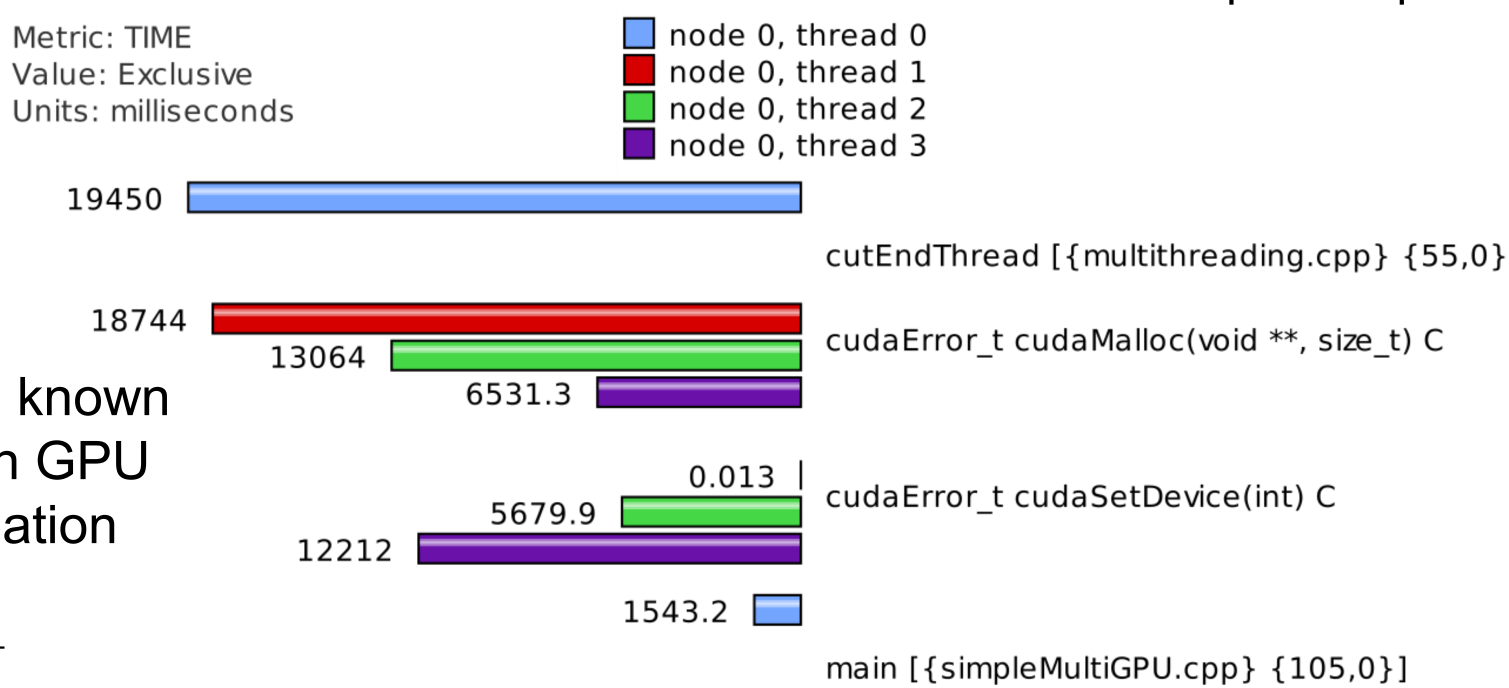
- Demonstration of multiple GPU device use
- *main* → *solverThread* → *reduceKernel*
- One Keeneland node with three GPUs
- Performance profile for:
  - One *main* thread
  - Three *solverThread* threads
  - Three *reduceKernel* “threads”

simpleMultiGPU Profile

Overall profile



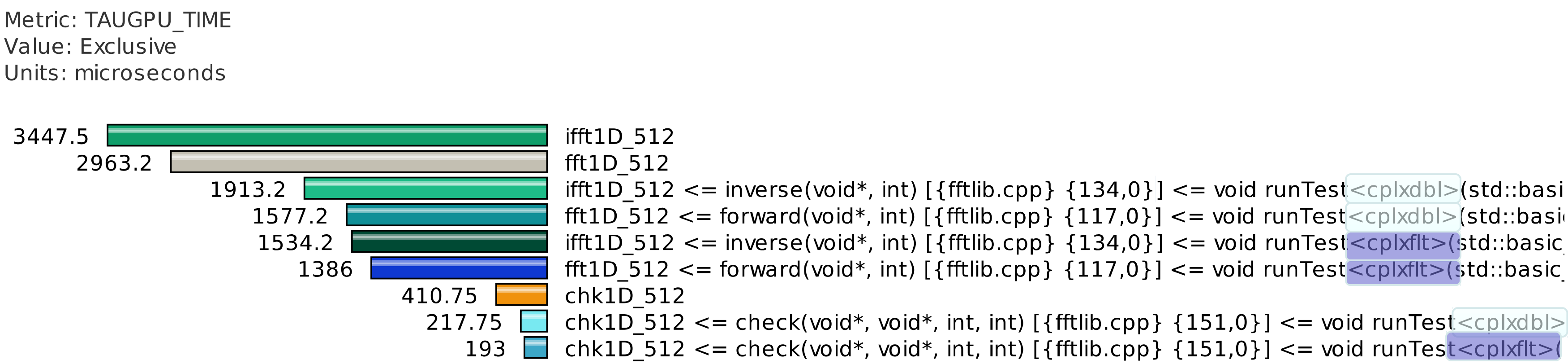
Comparison profile



Identified a known overhead in GPU context creation

SHOC FFT Profile with Callsite Info

- TAU is able to associate callsite context information with kernel launch so that different kernel calls can be distinguished

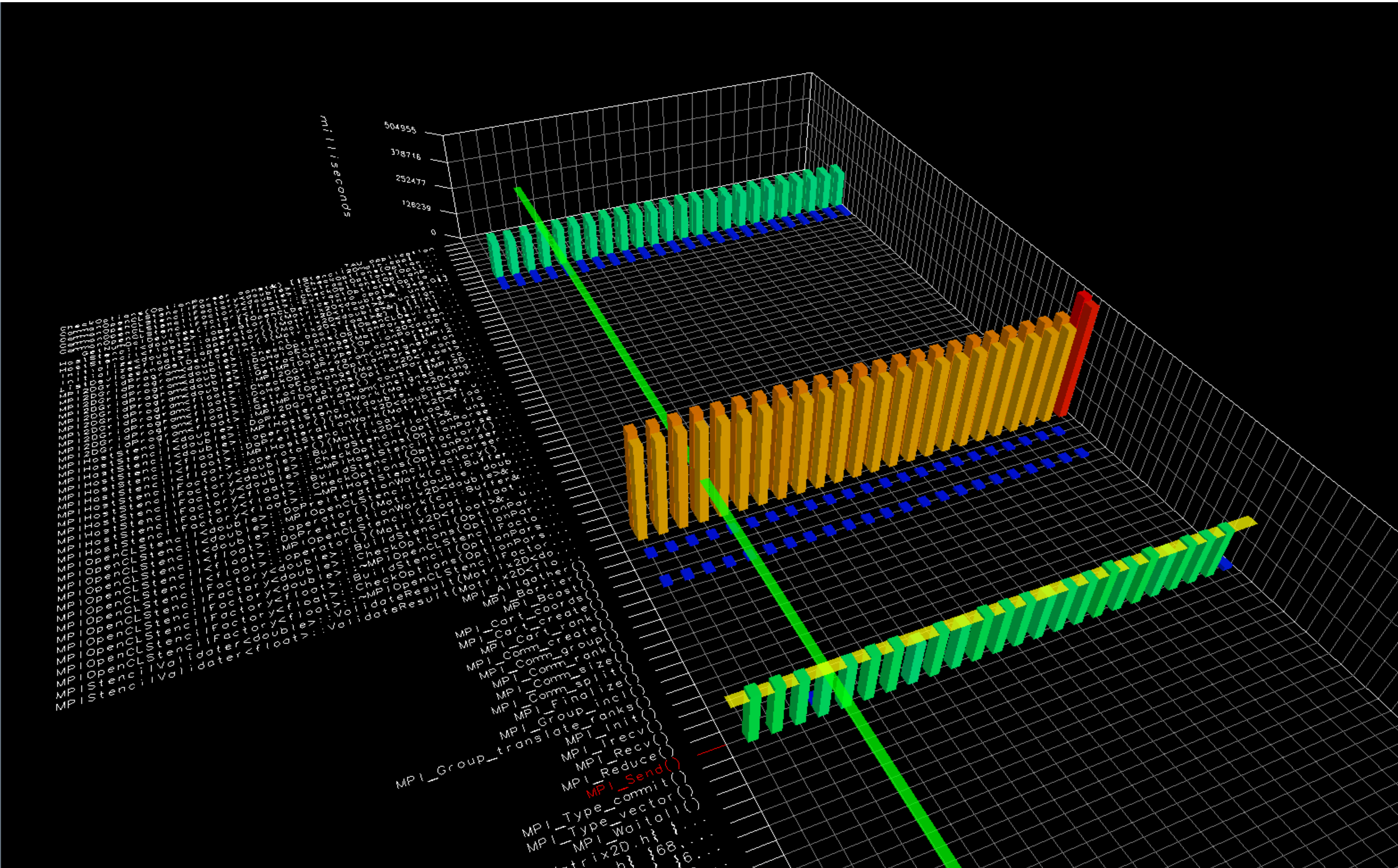


Each kernel (ifft1D_512, fft1D_512 and chk1D_512) is broken down by call-site, either during the single precession or double precession step.

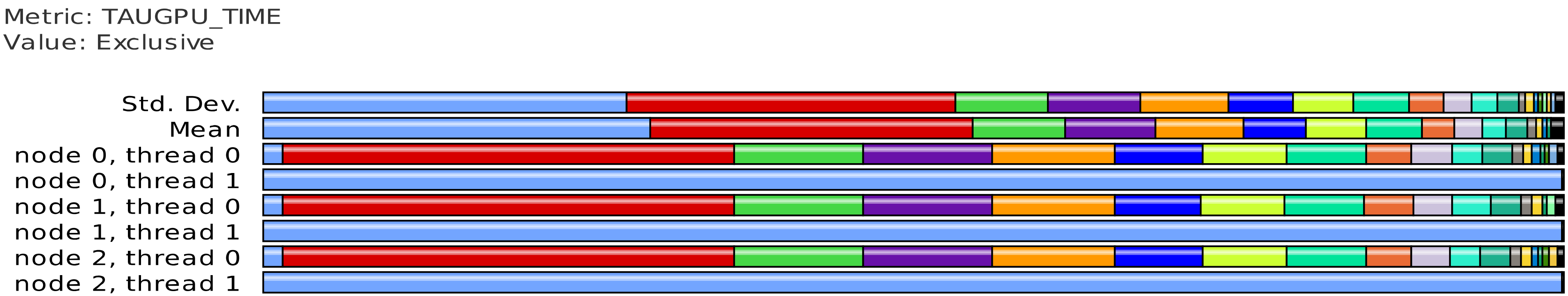
Example: SHOC Stencil2D

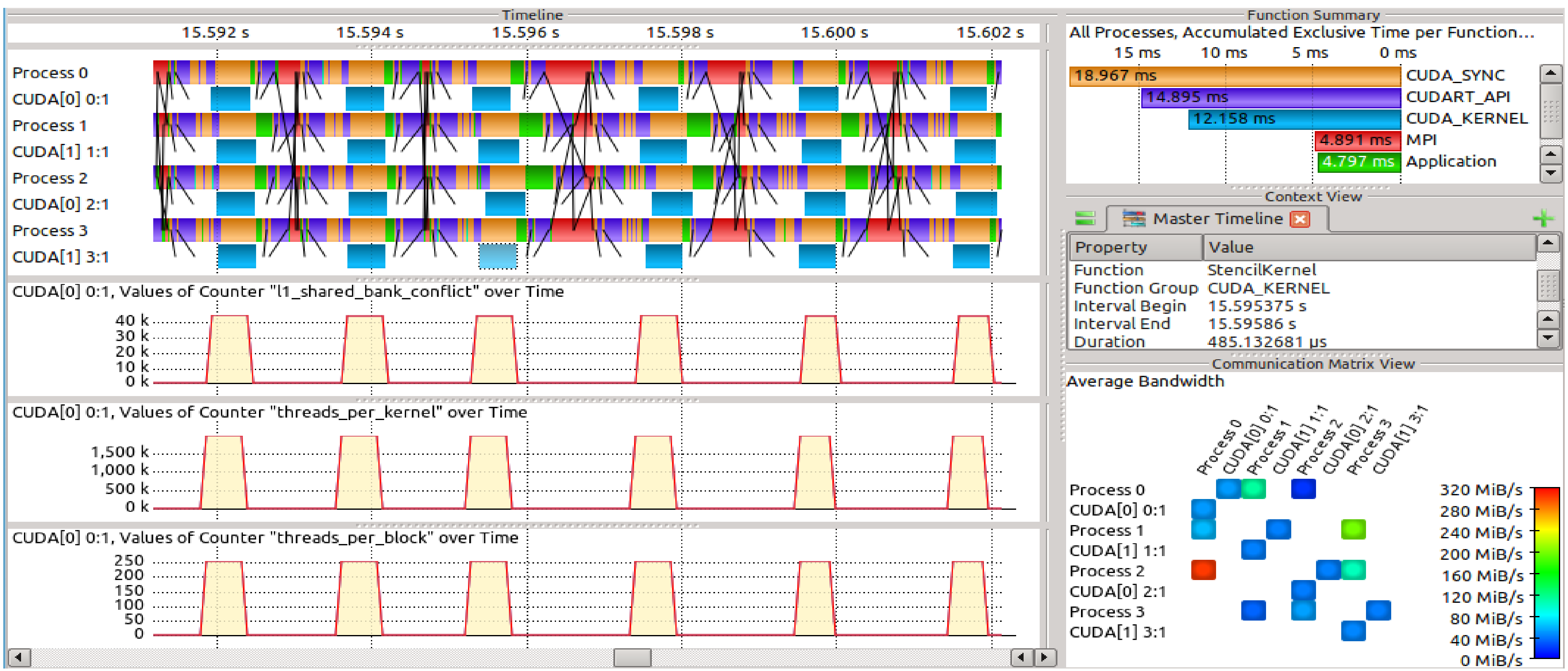
- Compute 2D, 9-point stencil
  - Multiple GPUs using MPI
  - CUDA and OpenCL versions
- One Keeneland node with 3 GPUs
- Eight Keeneland nodes with 24 GPUs
- Performance profile and trace
  - Application events
  - Communication events
  - Kernel execution

Stencil2D Parallel Profile

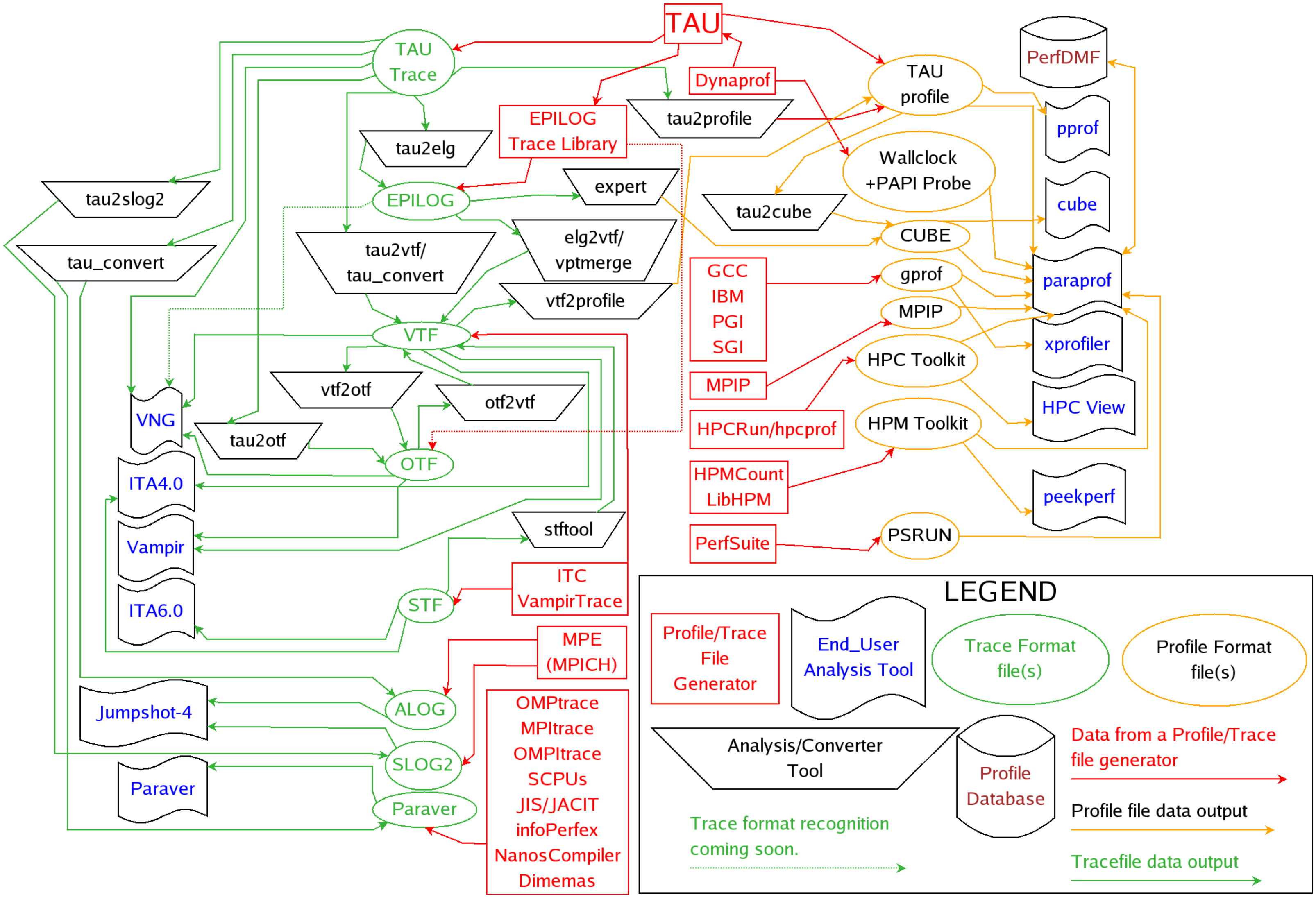


Stencil2D Parallel Profile / Trace in Vampir

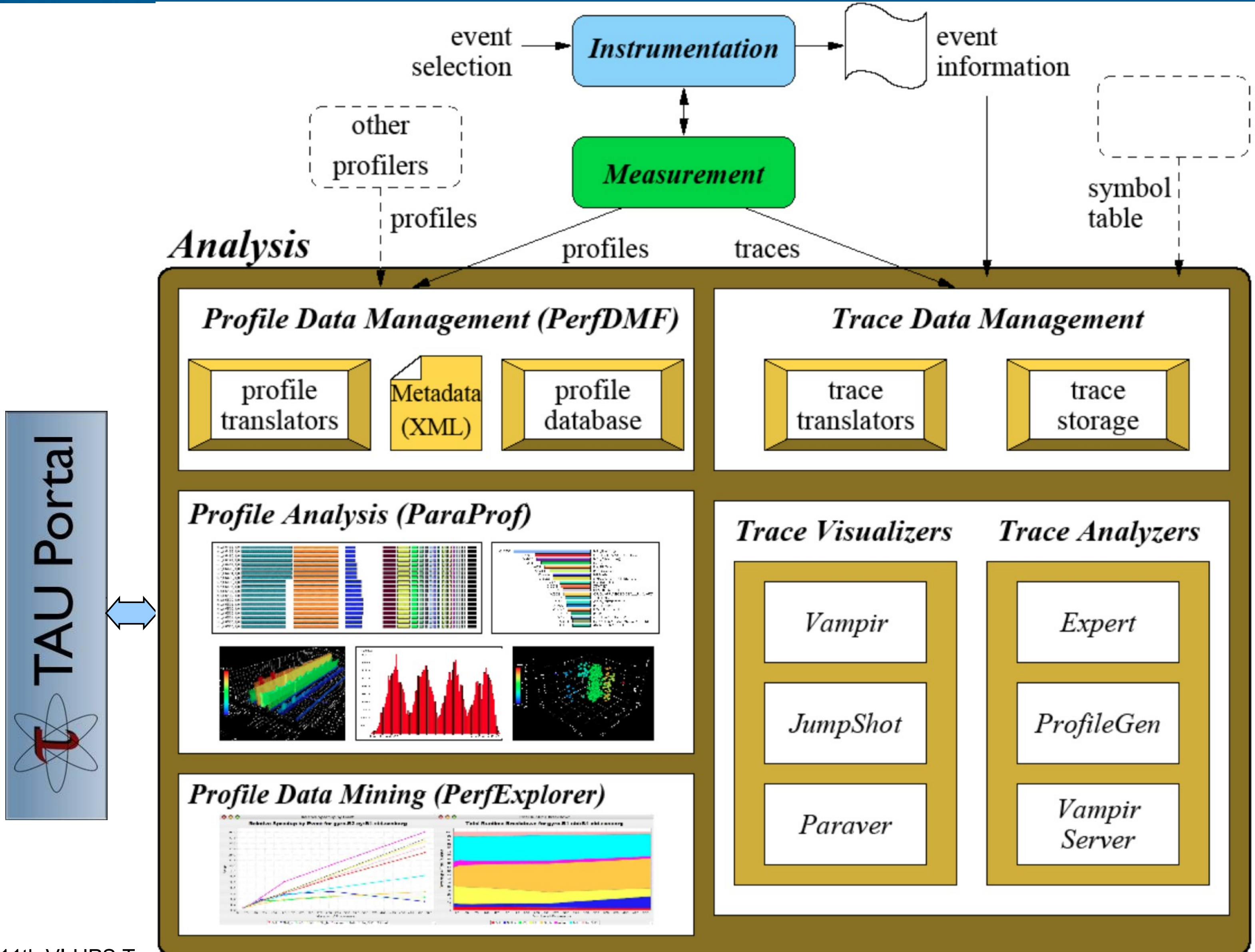




Building Bridges to Other Tools



TAU Analysis



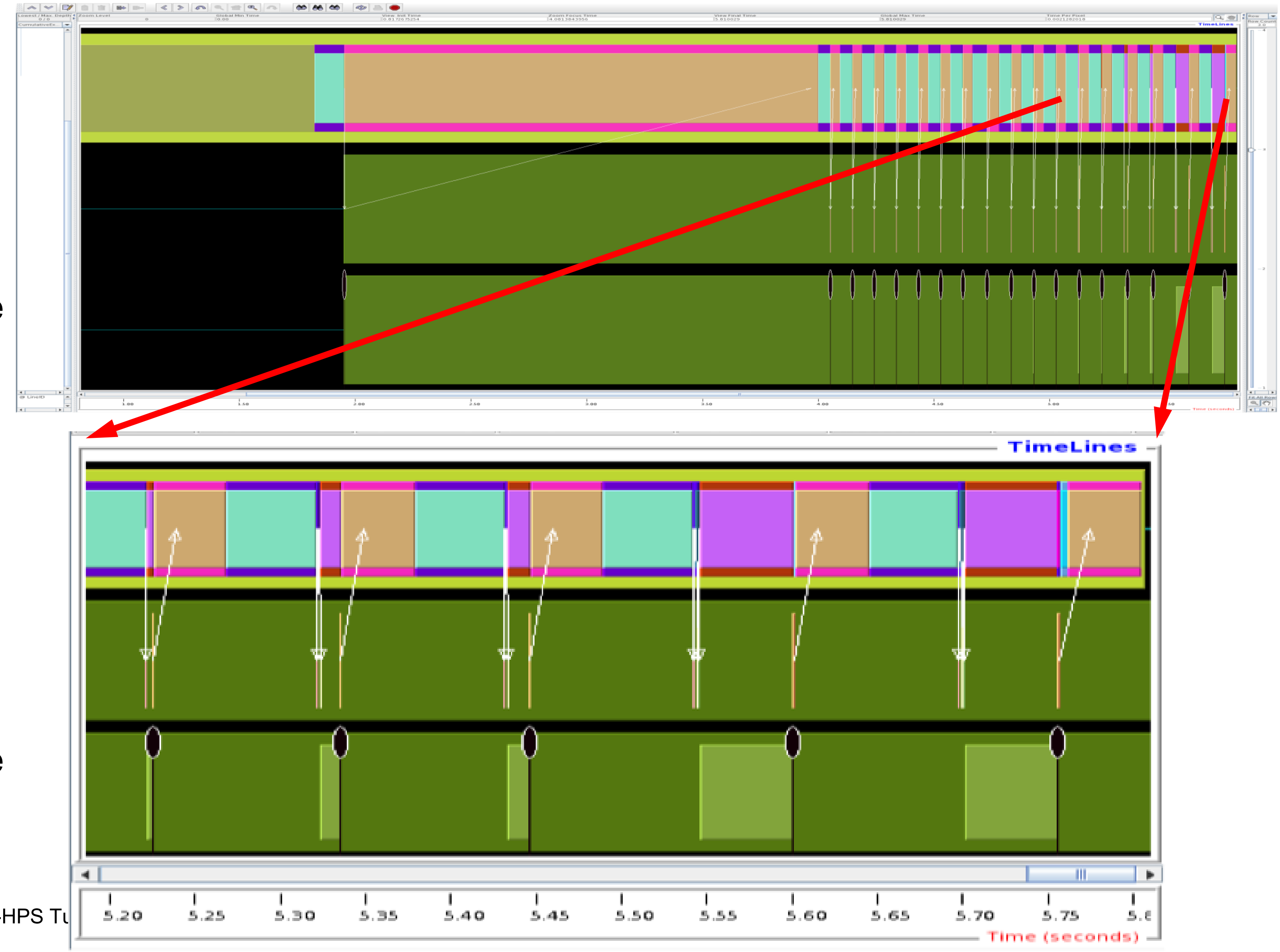
Example: NAMD with CUPTI

HMPP SGEMM (CAPS Entreprise)

Host
Process

Transfer
Kernel

Compute
Kernel



Host
Process

Transfer
Kernel

Compute
Kernel

Profiling PGI Accelerator Primitives

- PGI compiler allows users to annotate source code to identify loops that should be accelerated
- When a program is compiled with TAU, its measurement library intercepts the PGI runtime library layer to measure time spent in the runtime library routines and data transfers
- TAU also captures the arguments:
  - array data dimensions and sizes, strides, upload and download times, variable names, source file names, row and column information, and routines

Example: PGI GPU-accelerated MM

TAU: ParaProf: n,c,t 0,0,0 – /Users/sameer/rs/taudata/mm

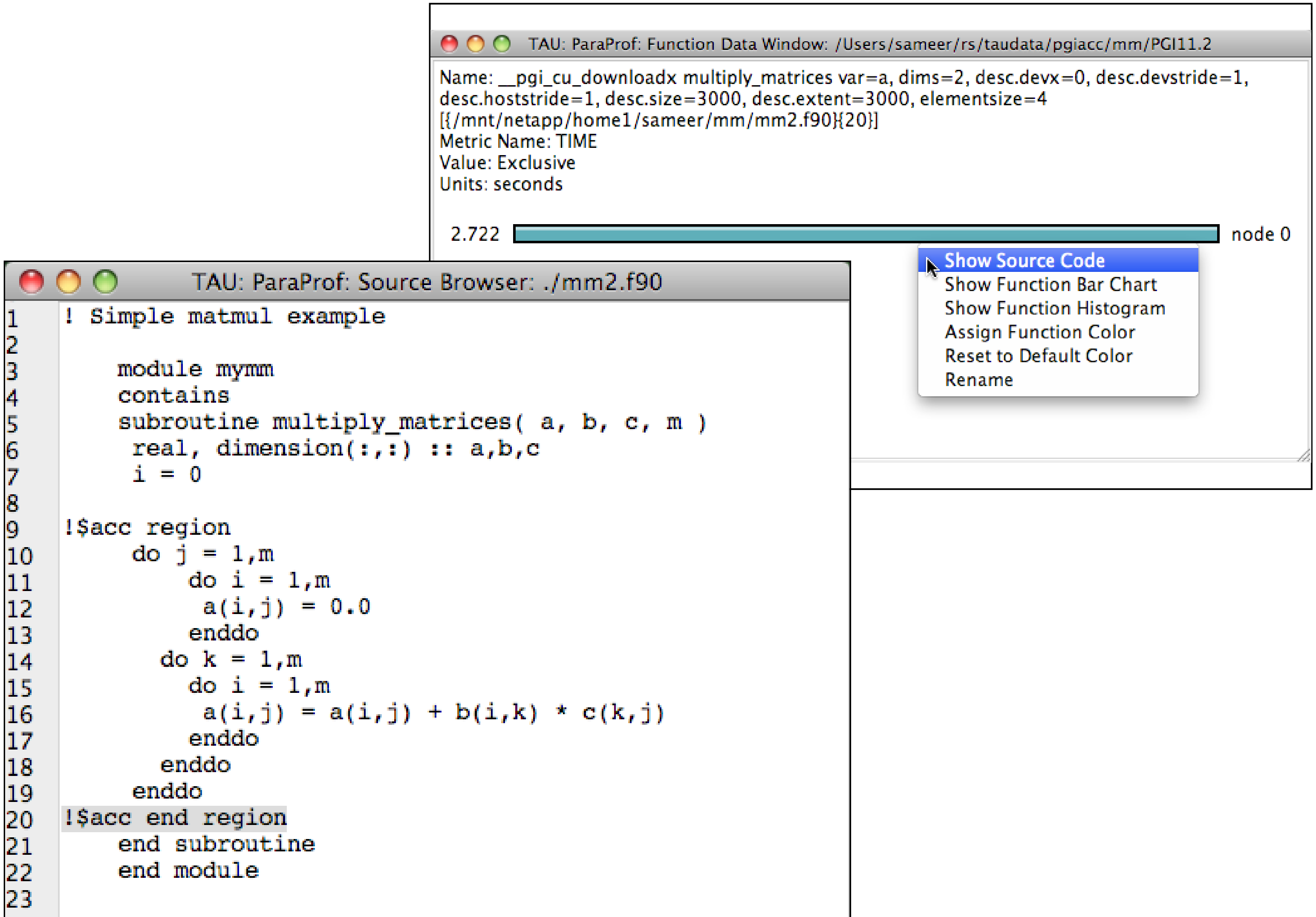
Metric: TIME
Value: Exclusive percent

55.367%	__pgi_cu_downloadx multiply_matrices var=a, dims=2, desc.devx=0, desc.devstride=1, desc.hoststride=1, desc.size=3000, desc.extent=
39.329%	__pgi_cu_init multiply_matrices [{/mnt/netapp/home1/sameer/mm/mm2.f90}{9}]
1.822%	mymatrixmultiply [{mmdriv.f90} {1,0}]
1.648%	__pgi_cu_uploadx multiply_matrices var=c, dims=2, desc.devx=0, desc.devstride=1, desc.hoststride=1, desc.size=3000, elementsize=4 [
1.618%	__pgi_cu_uploadx multiply_matrices var=b, dims=2, desc.devx=0, desc.devstride=1, desc.hoststride=1, desc.size=3000, elementsize=4 [
0.083%	__pgi_cu_free multiply_matrices [{/mnt/netapp/home1/sameer/mm/mm2.f90}]
0.07%	__pgi_cu_alloc multiply_matrices [{/mnt/netapp/home1/sameer/mm/mm2.f90}{9}]
0.037%	multiply_matrices [{mm2.f90} {5,0}]
0.007%	__pgi_cu_module multiply_matrices [{/mnt/netapp/home1/sameer/mm/mm2.f90}{9}]
0.006%	__pgi_cu_launch multiply_matrices (multiply_matrices_11_gpu,gx=188,gy=188,gz=1,bx=16,by=16,bz=1,flag=0) [{/mnt/netapp/home1/sa
0.005%	__pgi_cu_paramset multiply_matrices [{/mnt/netapp/home1/sameer/mm/mm2.f90}]
0.004%	__pgi_cu_launch multiply_matrices (multiply_matrices_15_gpu,gx=188,gy=188,gz=1,bx=16,by=16,bz=1,flag=0) [{/mnt/netapp/home1/sa
0.002%	__pgi_cu_module_function2 multiply_matrices name=multiply_matrices_11_gpu, argname=(null), argsize=20, varname=(null), varsize=0 [{/r
0.002%	__pgi_cu_module_function2 multiply_matrices name=multiply_matrices_15_gpu, argname=(null), argsize=44, varname=(null), varsize=0 [{/r

TAU: ParaProf: Thread Statistics: n,c,t, 0,0,0 – /Users/sameer/rs/taudata/mm

Exclusive TIME %	Name	Exclusive TIME ▽	Inclusive TIME	Calls	Child Calls
55.4%	__pgi_cu_downloadx multiply_matrices var=a, dims=2, desc.devx=0, desc.devstride=1, desc.hostst	2.721	2.721	5	0
39.3%	__pgi_cu_init multiply_matrices [{/mnt/netapp/home1/sameer/mm/mm2.f90}{9}]	1.933	1.933	5	0
1.8%	mymatrixmultiply [{mmdriv.f90} {1,0}]	0.09	4.914	1	5
1.6%	__pgi_cu_uploadx multiply_matrices var=c, dims=2, desc.devx=0, desc.devstride=1, desc.hoststrid	0.081	0.081	5	0
1.6%	__pgi_cu_uploadx multiply_matrices var=b, dims=2, desc.devx=0, desc.devstride=1, desc.hoststrid	0.079	0.079	5	0
0.1%	__pgi_cu_free multiply_matrices [{/mnt/netapp/home1/sameer/mm/mm2.f90}]	0.004	0.004	15	0
0.1%	__pgi_cu_alloc multiply_matrices [{/mnt/netapp/home1/sameer/mm/mm2.f90}{9}]	0.003	0.003	15	0
0.0%	multiply_matrices [{mm2.f90} {5,0}]	0.002	4.825	5	85
0.0%	__pgi_cu_module multiply_matrices [{/mnt/netapp/home1/sameer/mm/mm2.f90}{9}]	0	0	5	0
0.0%	__pgi_cu_launch multiply_matrices (multiply_matrices_11_gpu,gx=188,gy=188,gz=1,bx=16,by=16	0	0	5	0
0.0%	__pgi_cu_paramset multiply_matrices [{/mnt/netapp/home1/sameer/mm/mm2.f90}]	0	0	10	0
0.0%	__pgi_cu_launch multiply_matrices (multiply_matrices_15_gpu,gx=188,gy=188,gz=1,bx=16,by=16	0	0	5	0
0.0%	__pgi_cu_module_function2 multiply_matrices name=multiply_matrices_11_gpu, argname=(null), arg	0	0	5	0

PGI MM Computational Kernel



Instrumentation: Re-writing Binaries

- Support for both static and dynamic executables
- Specify the list of routines to instrument/exclude from instrumentation
- Specify the TAU measurement library to be injected
- Simplify the usage of TAU:
  - To instrument:

    ```
    % tau_run a.out -o a.inst
    ```
  - To perform measurements, execute the application:

    ```
    % mpirun -np 8 ./a.inst
    ```
  - To analyze the data:

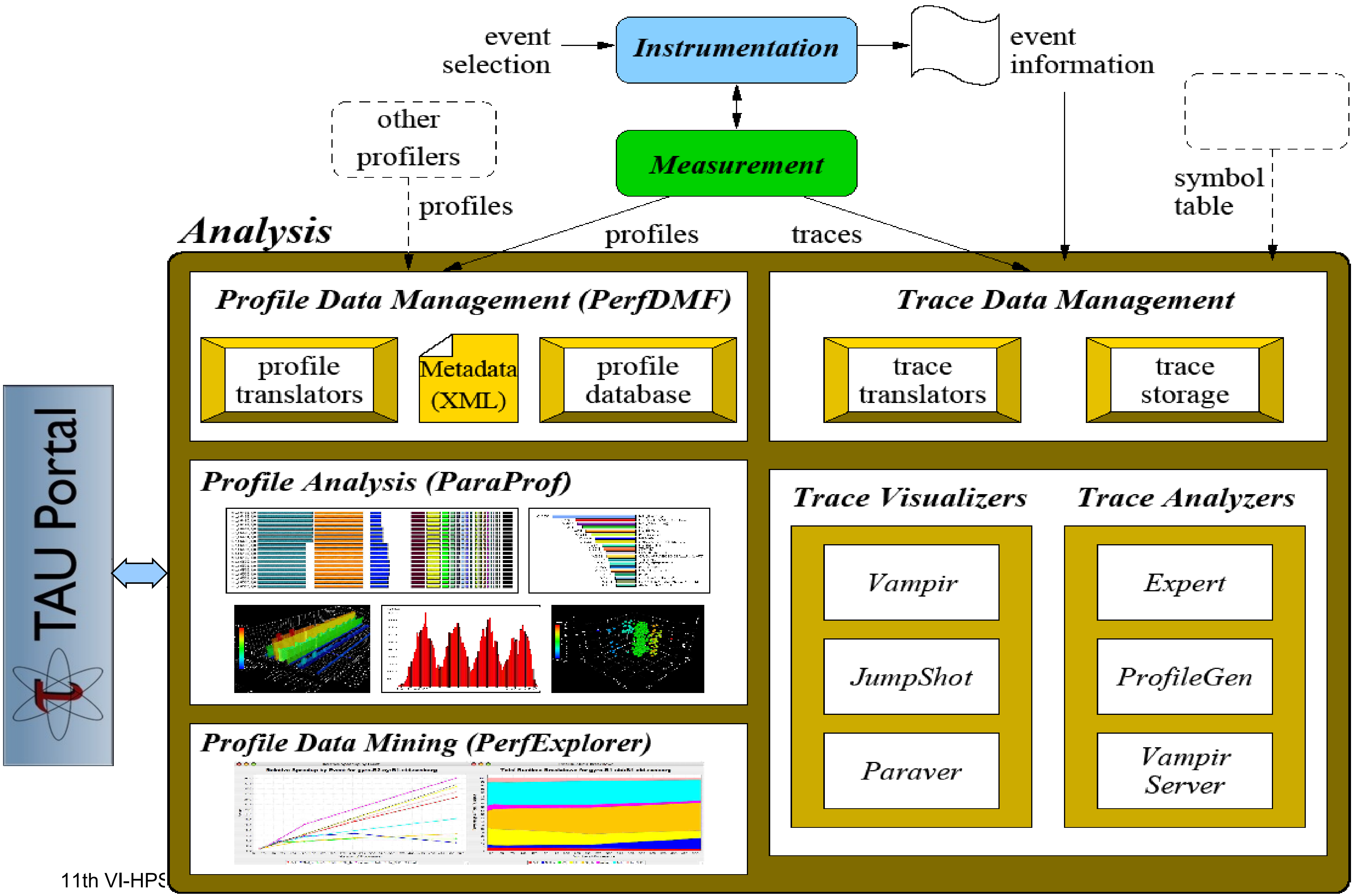
    ```
    % paraprof
    ```

tau_run with NAS PBS

livetau@paratools01:~

```
/home/livetau% cd ~/tutorial
/home/livetau/tutorial% # Build an uninstrumented bt NAS Parallel Benchmark
/home/livetau/tutorial% make bt CLASS=W NPROCS=4
/home/livetau/tutorial% cd bin
/home/livetau/tutorial/bin% # Run the instrumented code
/home/livetau/tutorial/bin% mpirun -np 4 ./bt_W.4
/home/livetau/tutorial/bin%
/home/livetau/tutorial/bin% # Instrument the executable using TAU with DyninstAPI
/home/livetau/tutorial/bin%
/home/livetau/tutorial/bin% tau_run ./bt_W.4 -o ./bt.i
/home/livetau/tutorial/bin% rm -rf profile.* MULT*
/home/livetau/tutorial/bin% mpirun -np 4 ./bt.i
/home/livetau/tutorial/bin% paraprof
/home/livetau/tutorial/bin%
/home/livetau/tutorial/bin% # Choose a different TAU configuration
/home/livetau/tutorial/bin% ls $TAU/libTAUsh
libTAUsh-depthlimit-mpi-pdt.so*                 libTAUsh-papi-pdt.so*
libTAUsh-mpi-pdt.so*                            libTAUsh-papi-pthread-pdt.so*
libTAUsh-mpi-pdt-upc.so*                        libTAUsh-param-mpi-pdt.so*
libTAUsh-mpi-python-pdt.so*                     libTAUsh-pdt.so*
libTAUsh-papi-mpi-pdt.so*                       libTAUsh-pdt-trace.so*
libTAUsh-papi-mpi-pdt-upc.so*                   libTAUsh-phase-papi-mpi-pdt.so*
libTAUsh-papi-mpi-pdt-upc-udp.so*               libTAUsh-pthread-pdt.so*
libTAUsh-papi-mpi-pdt-vampirtrace-trace.so* libTAUsh-python-pdt.so*
libTAUsh-papi-mpi-python-pdt.so*
/home/livetau/tutorial/bin%
/home/livetau/tutorial/bin% tau_run -XrunTAUsh-papi-mpi-pdt-vampirtrace-trace bt_W.4 -o bt.vpt
/home/livetau/tutorial/bin% setenv VT_METRICS PAPI_FP_INS:PAPI_L1_DCM
/home/livetau/tutorial/bin% mpirun -np 4 ./bt.vpt
/home/livetau/tutorial/bin% vampir bt.vpt.otf &
```

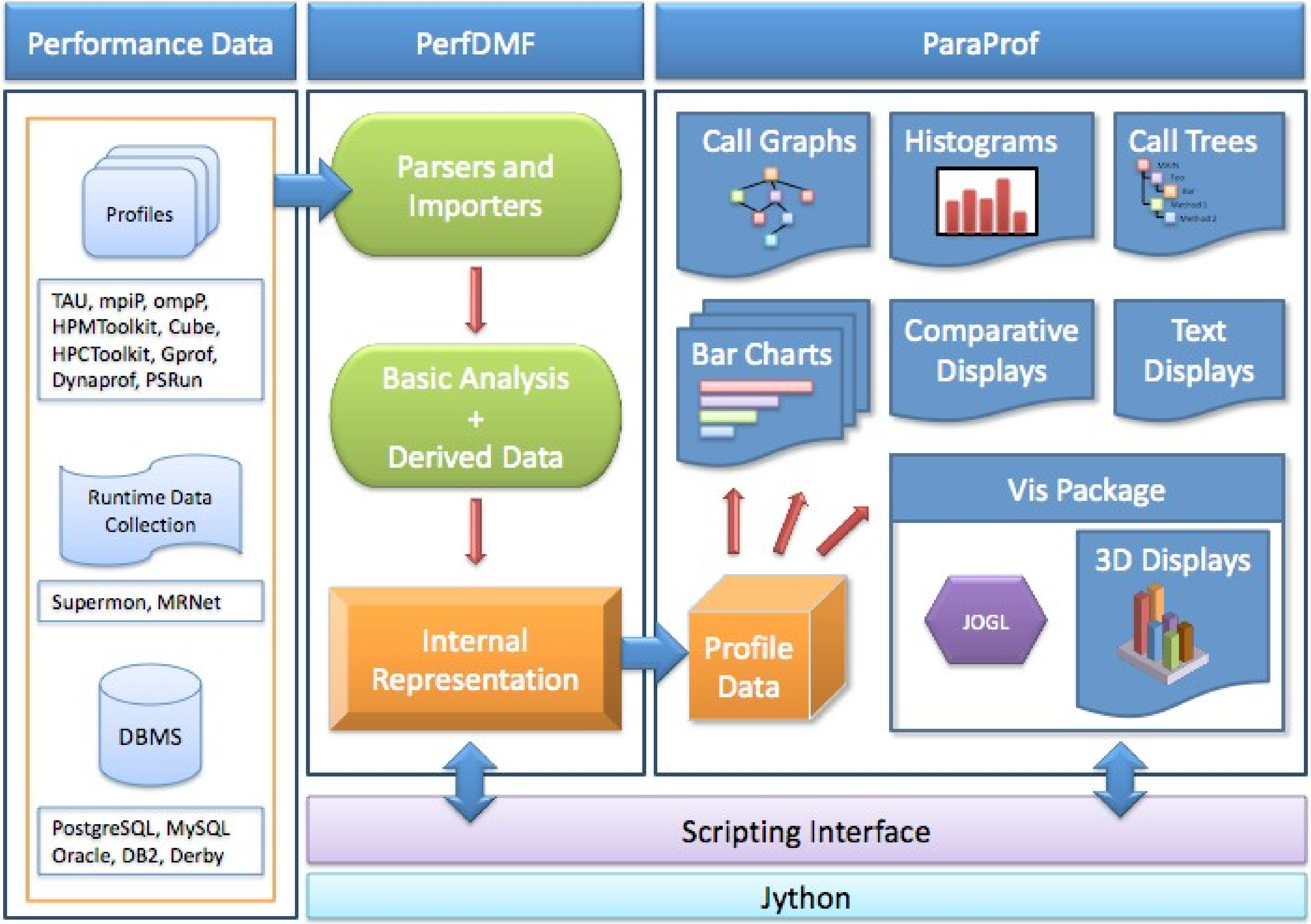
TAU Analysis



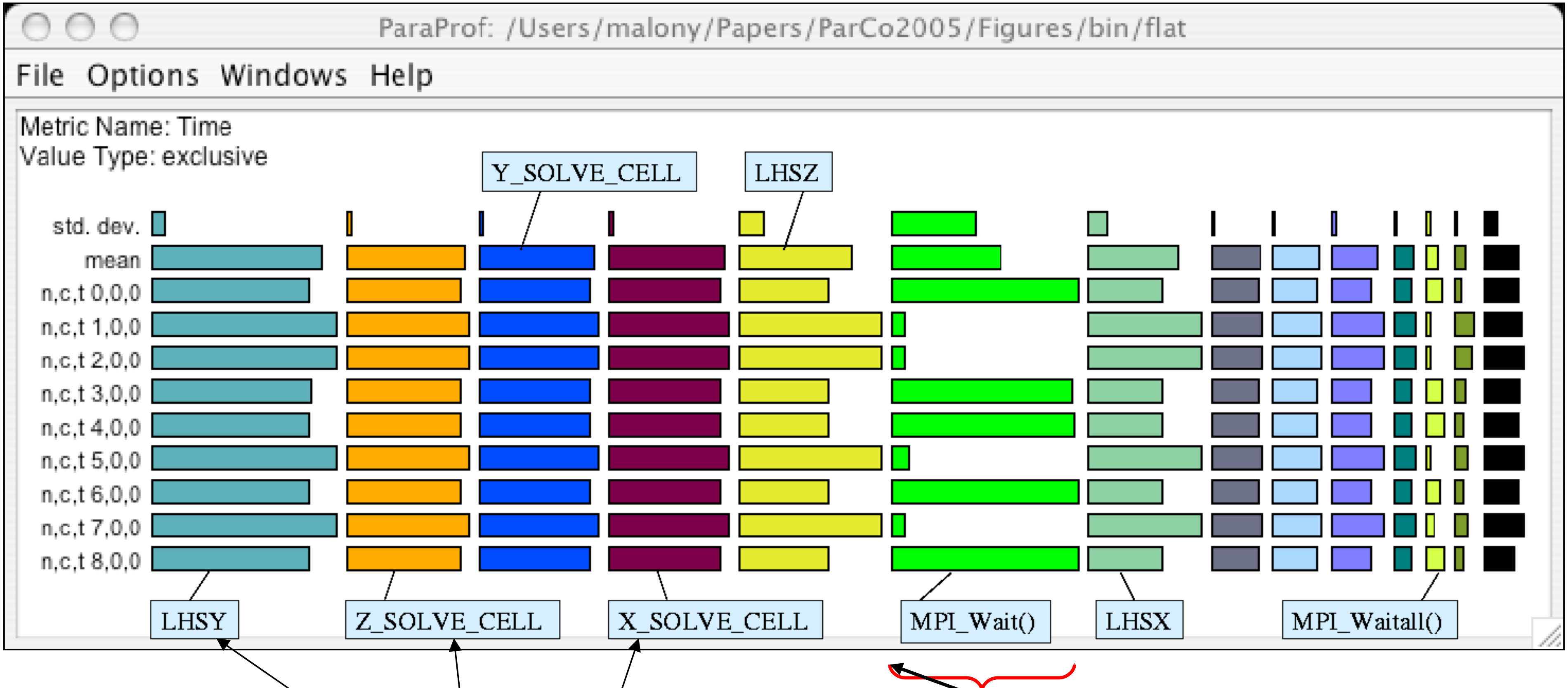
Performance Analysis

- Analysis of parallel profile and trace measurement
- Parallel profile analysis (ParaProf)
  - Java-based analysis and visualization tool
  - Support for large-scale parallel profiles
- Performance data management framework (PerfDMF)
- Parallel trace analysis
  - Translation to VTF (V3.0), EPILOG, OTF formats
  - Integration with Vampir / Vampir Server (TU Dresden)
  - Profile generation from trace data
- Online parallel analysis and visualization
- Integration with CUBE browser (Scalasca, UTK / FZJ)

ParaProf Profile Analysis Framework



NAS BT – Flat Profile



Application routine names reflect phase semantics

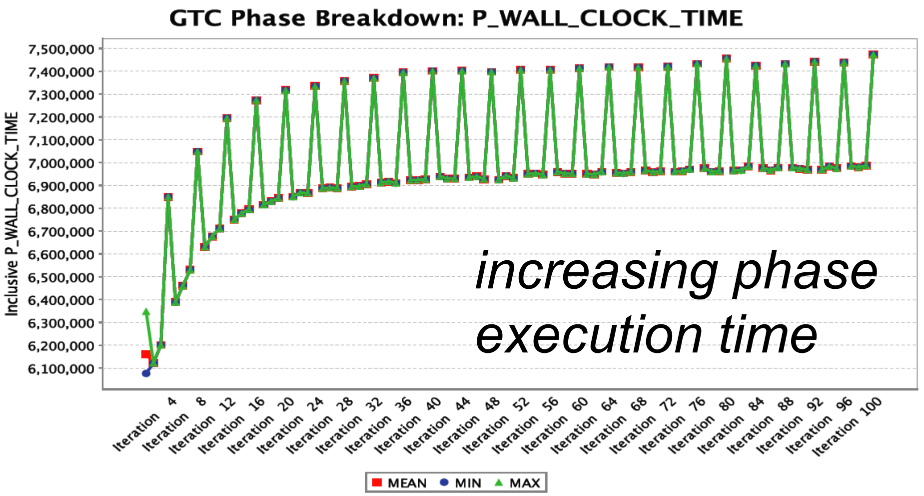
How is MPI_Wait() distributed relative to solver direction?

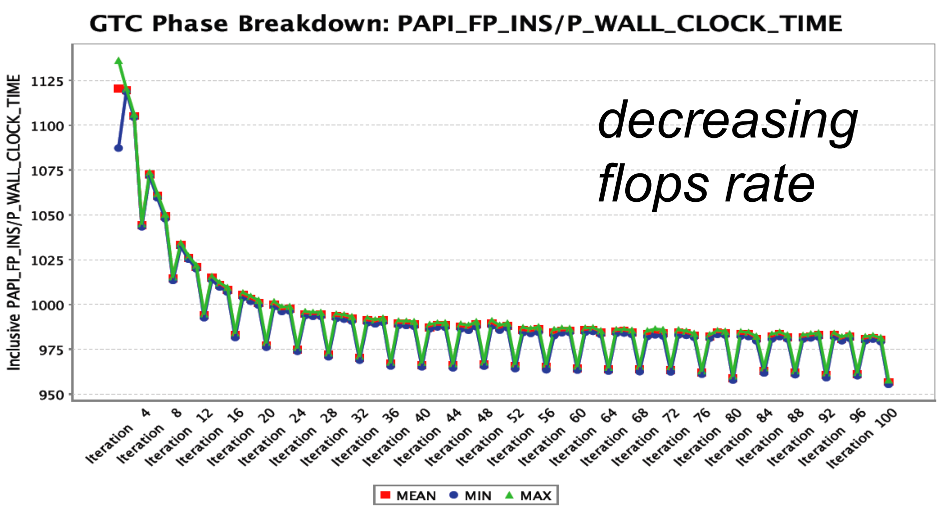
NAS BT – Phase Profile

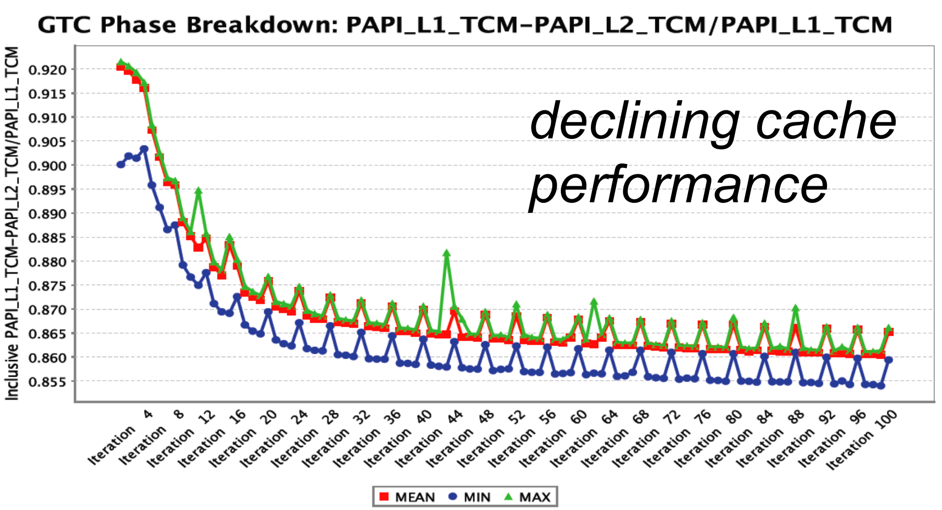
Main phase shows nested phases and immediate events

Phase Profiling of HW Counters

- GTC particle-in-cell simulation of fusion turbulence
- Phases assigned to iterations
- Poor temporal locality for one important data
- Automatically generated by PE2 python script

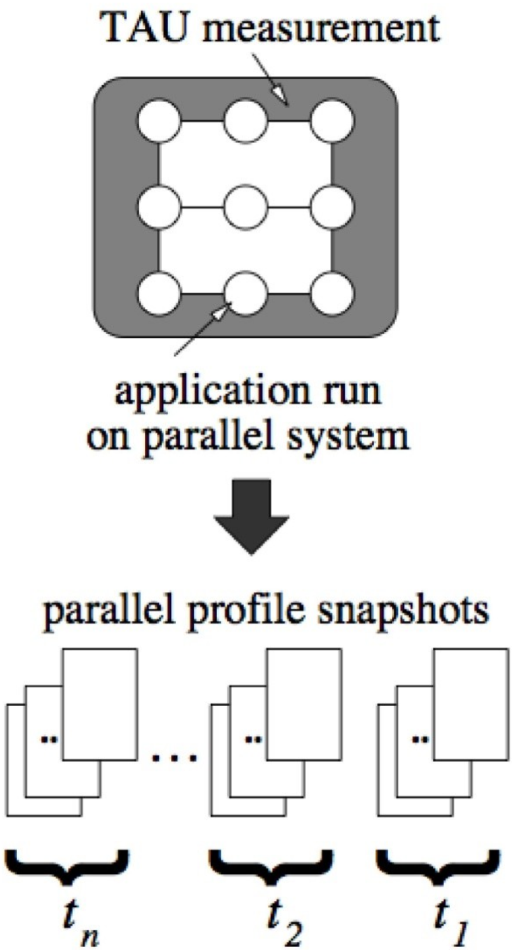


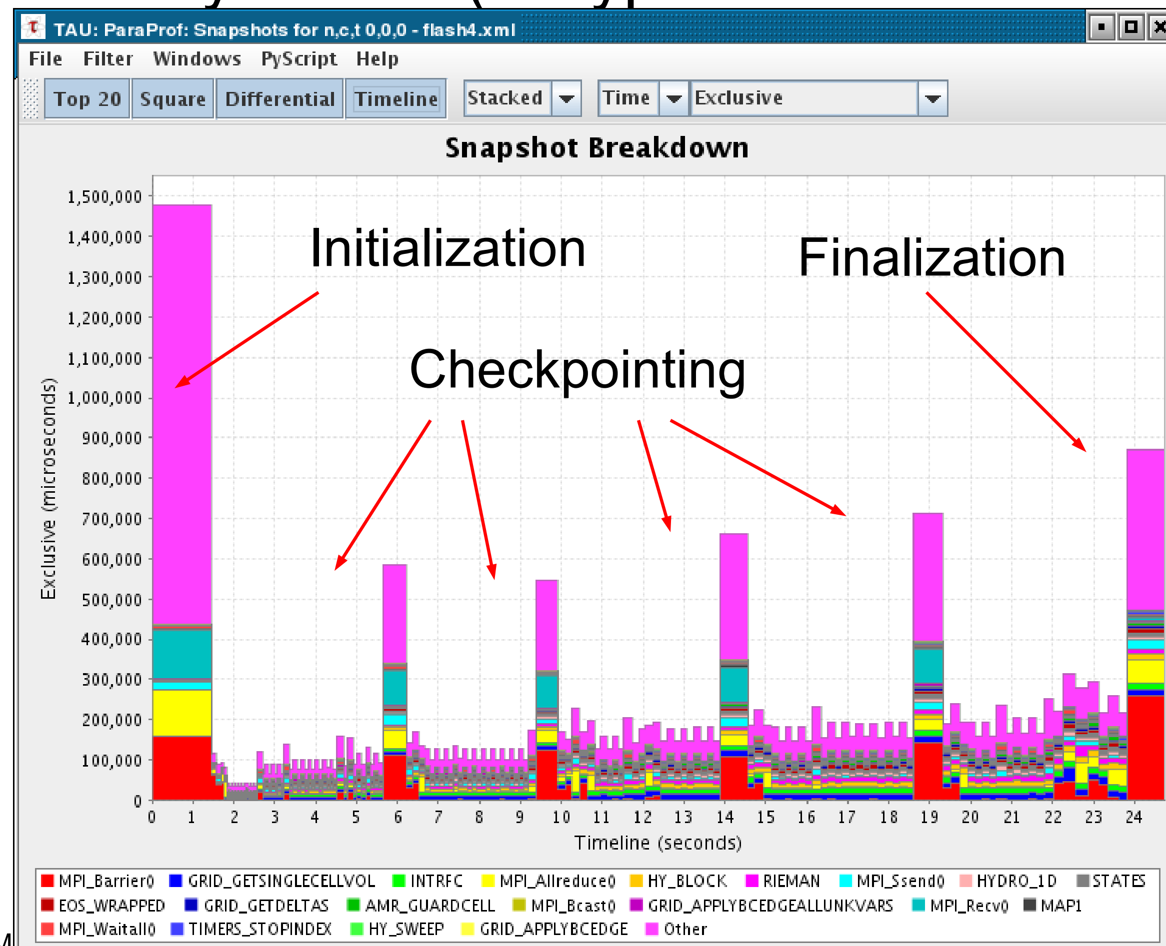




Profile Snapshots in ParaProf

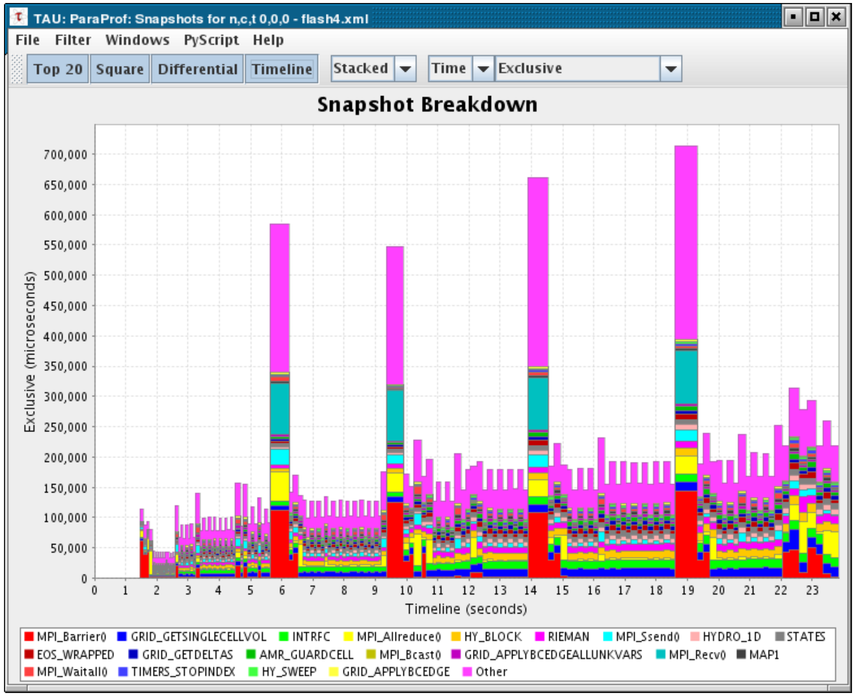
- Profile snapshots are parallel profiles recorded at runtime
- Shows performance profile dynamics (all types

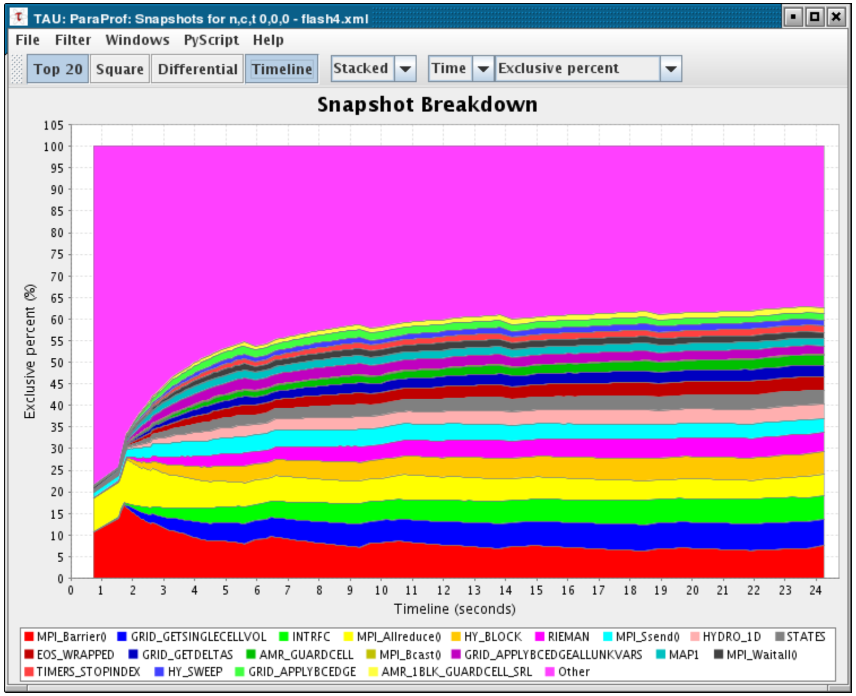




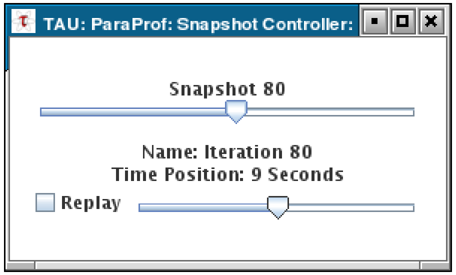
Profile Snapshot Views

- Percentage breakdown
- Only show main loop

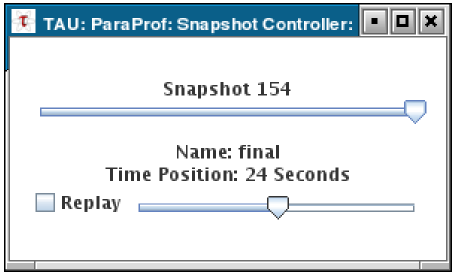


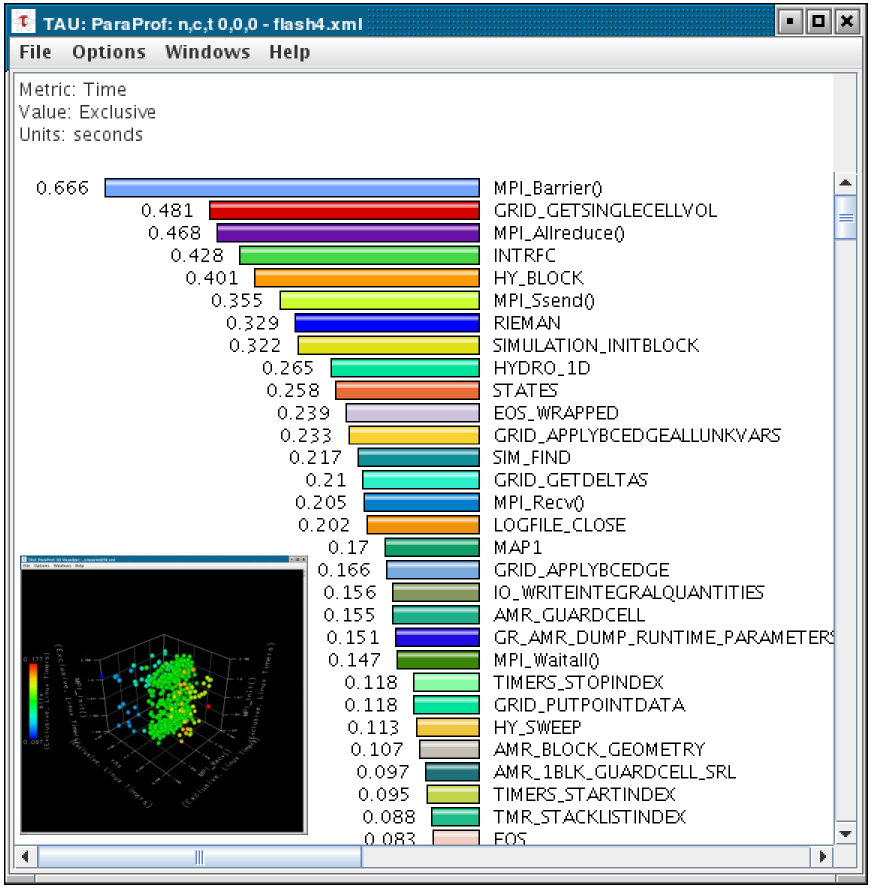


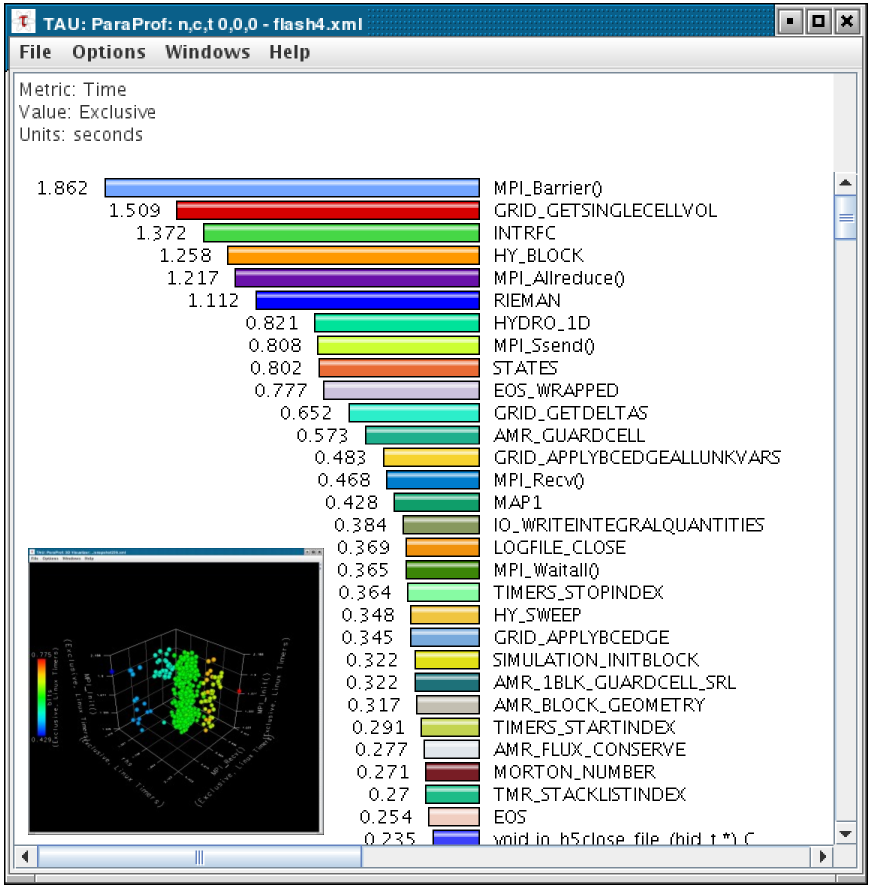
Snapshot Replay in ParaProf



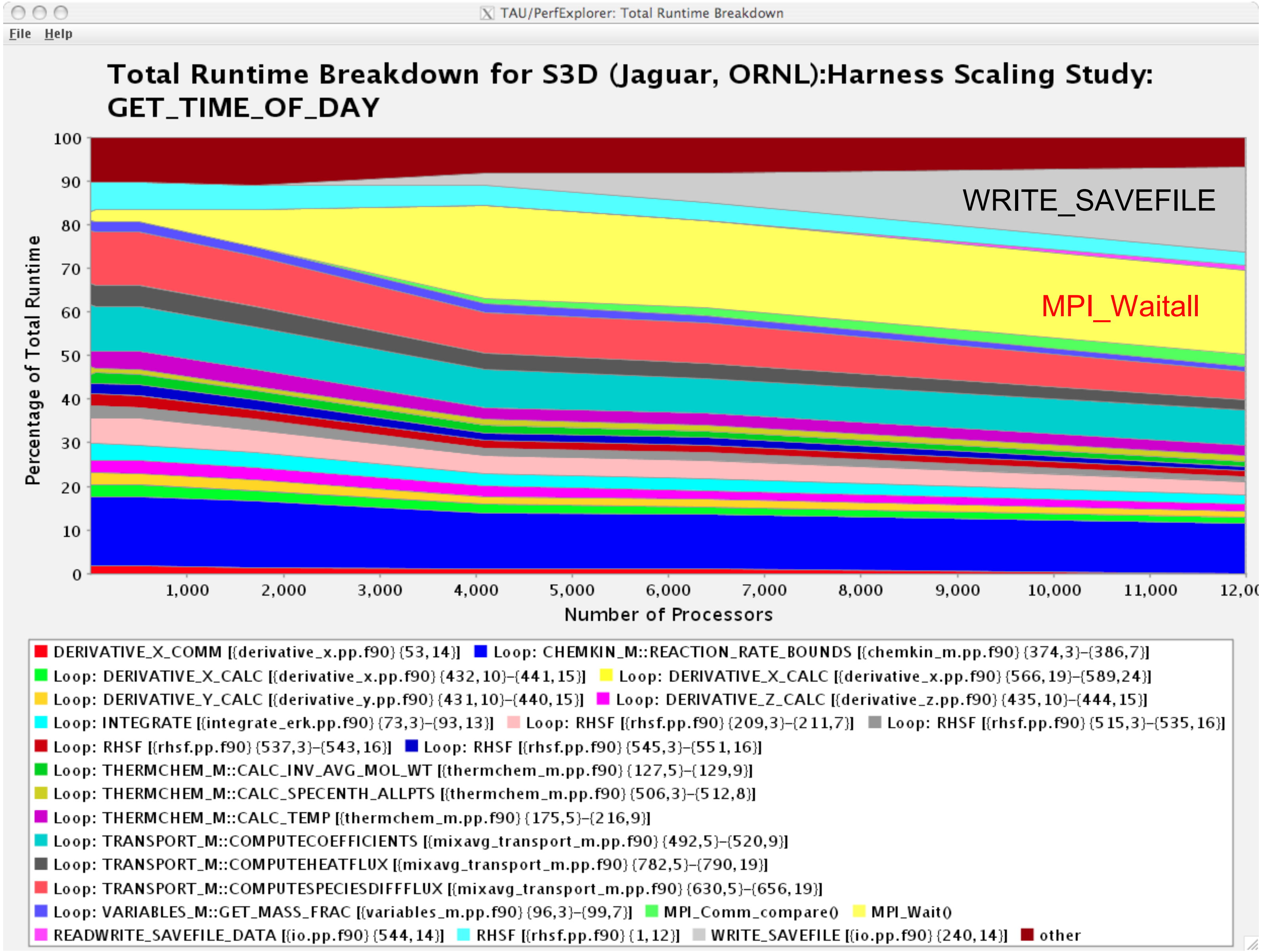
All windows dynamically update





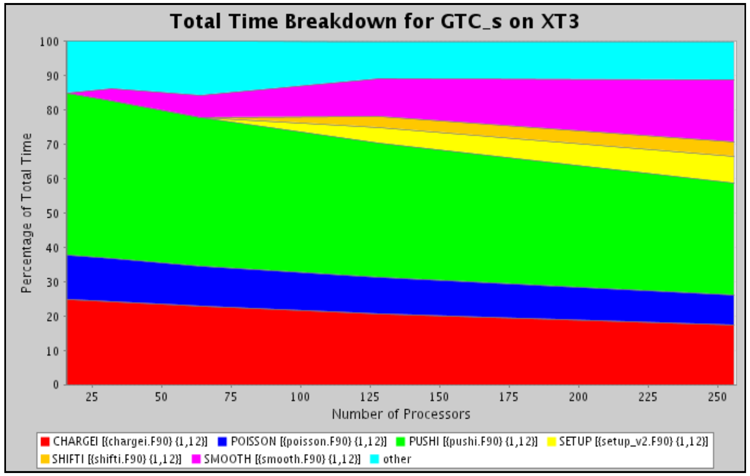


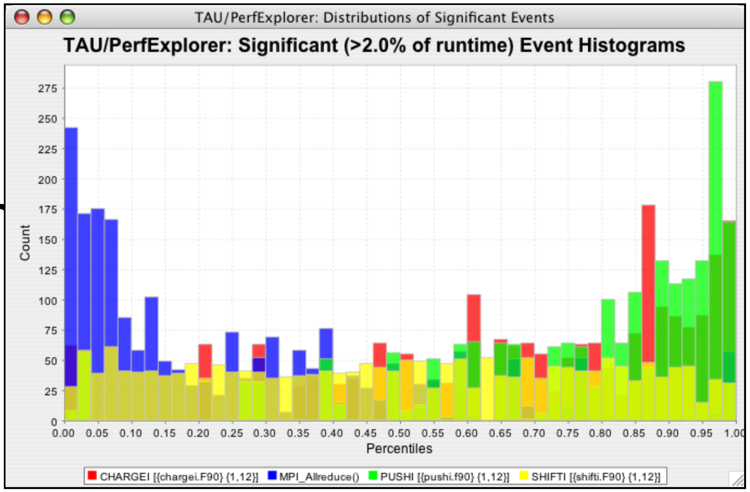
PerfExplorer – Runtime Breakdown



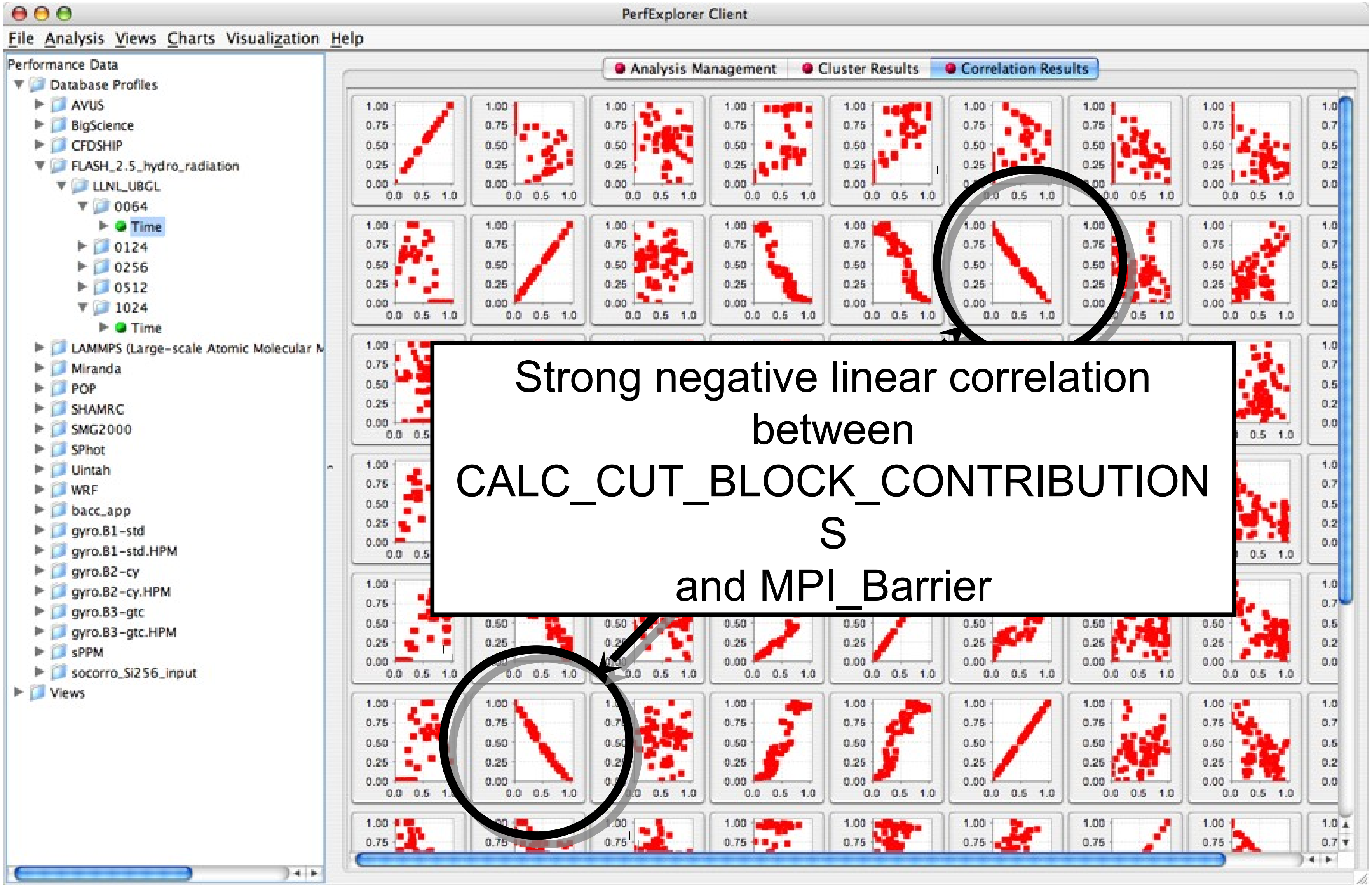
PerfExplorer – Relative Comparisons

- Total execution time
- Timesteps per second
- Relative efficiency
- Relative efficiency per event
- Relative speedup
- Relative speedup per event
- Group fraction of total
- Runtime breakdown
- Correlate events with total runtim
- Relative efficiency per phase
- Relative speedup per phase
- Distribution visualizations



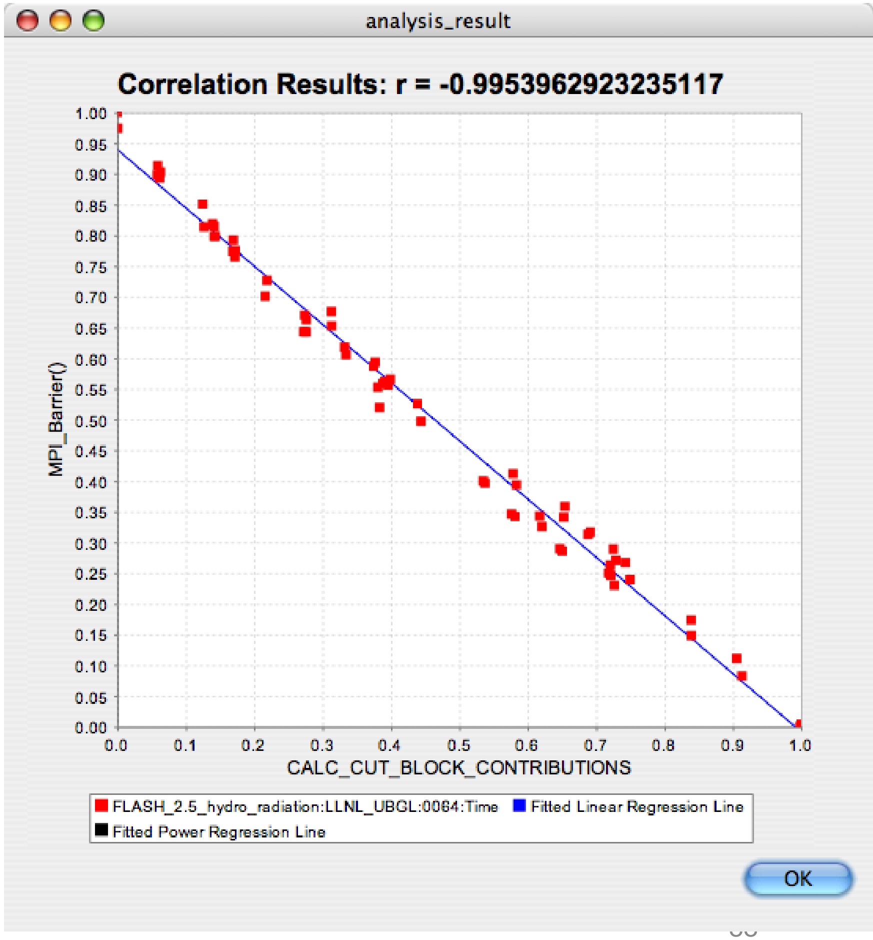


PerfExplorer – Correlation Analysis



PerfExplorer – Correlation Analysis

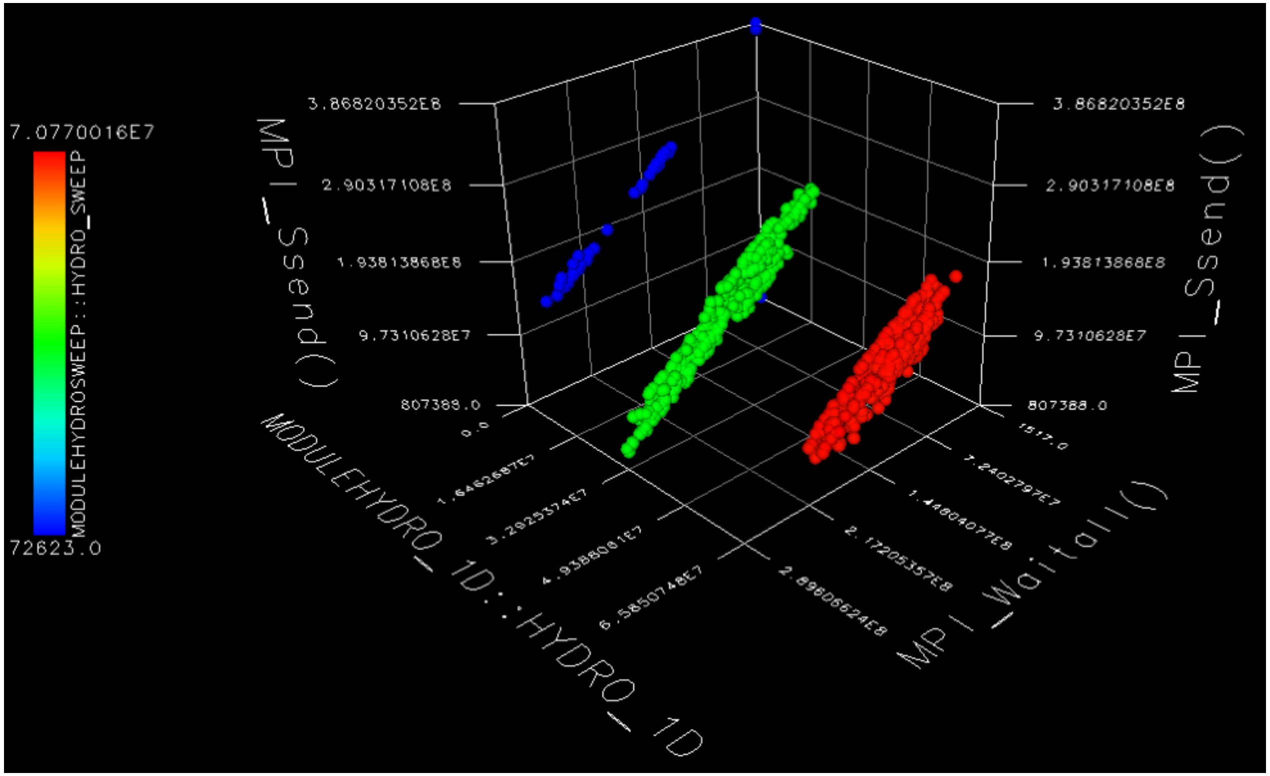
- -0.995 indicates strong, negative relationship
- As CALC_CUT_ BLOCK_CONTRIBUTIONS() increases in execution time, MPI_Barrier() decreases



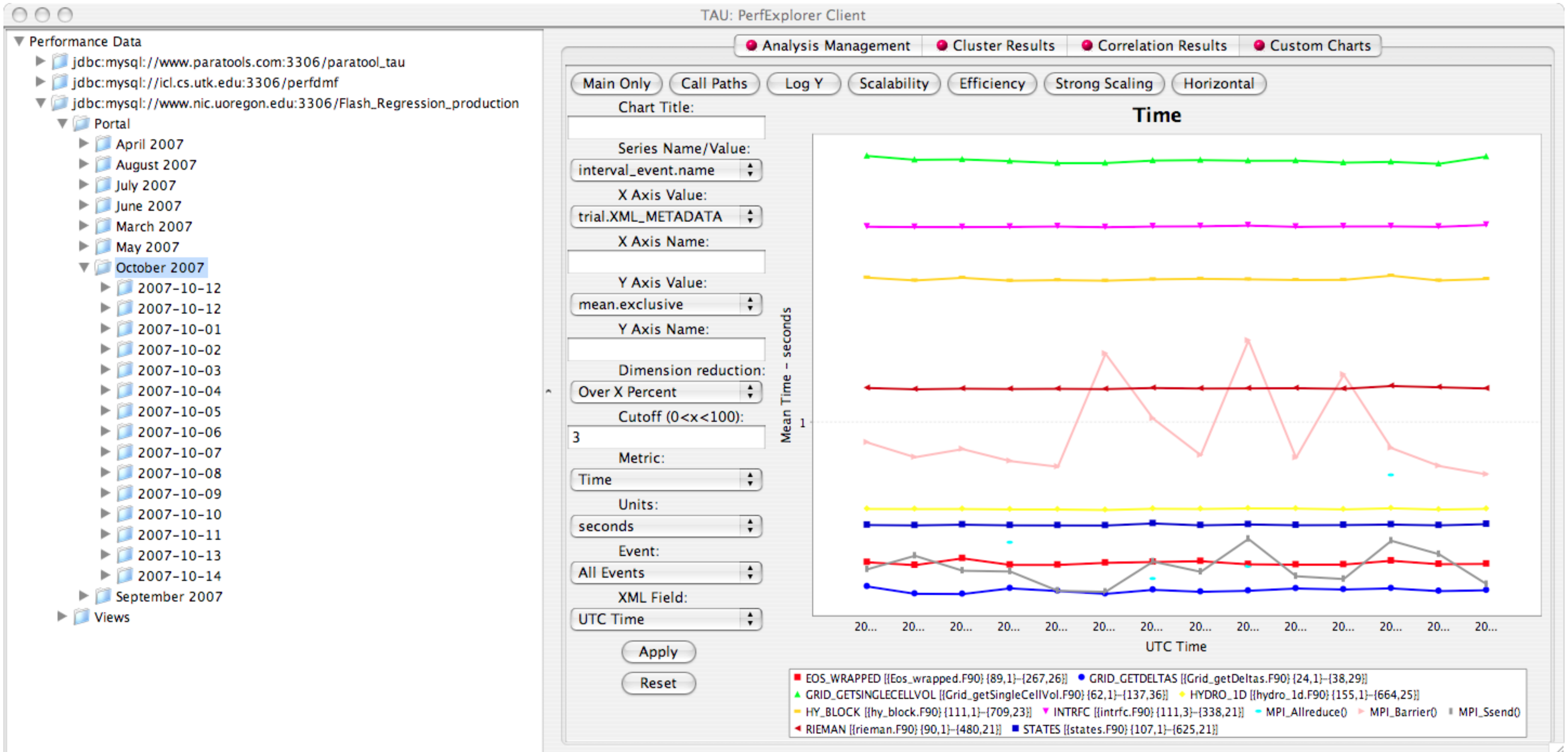
PerfExplorer – Cluster Analysis

PerfExplorer – Cluster Analysis

- Four significant events automatically selected
- Clusters and correlations are visible

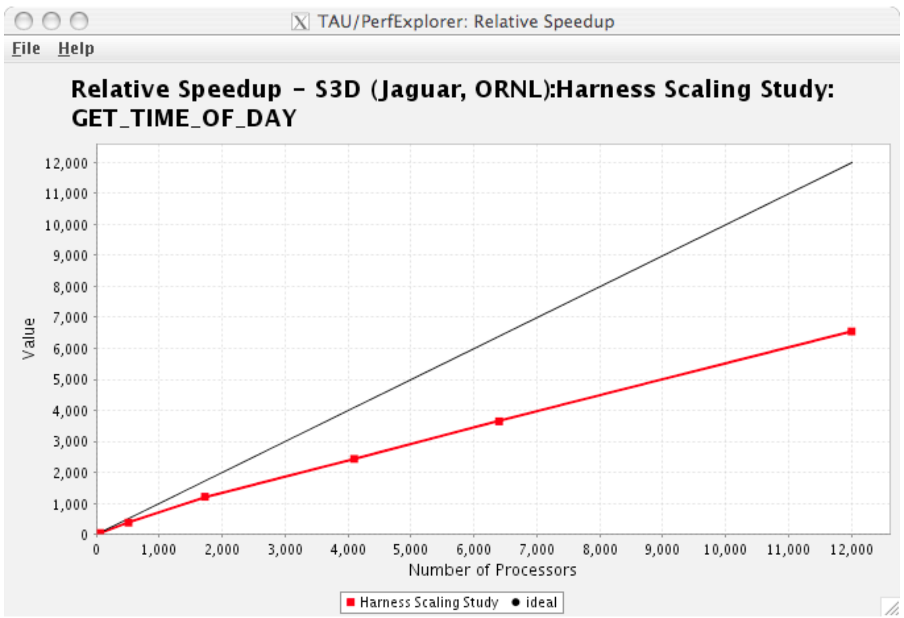


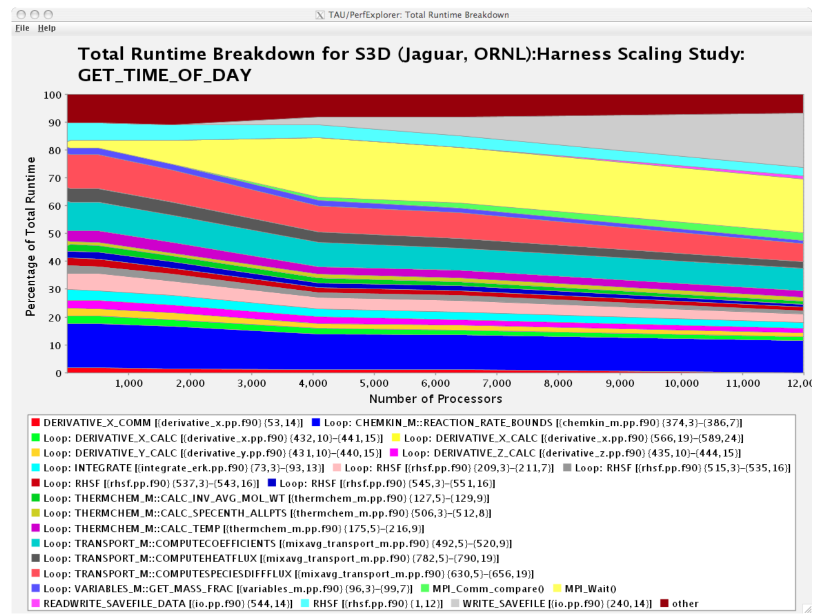
PerfExplorer – Performance Regression



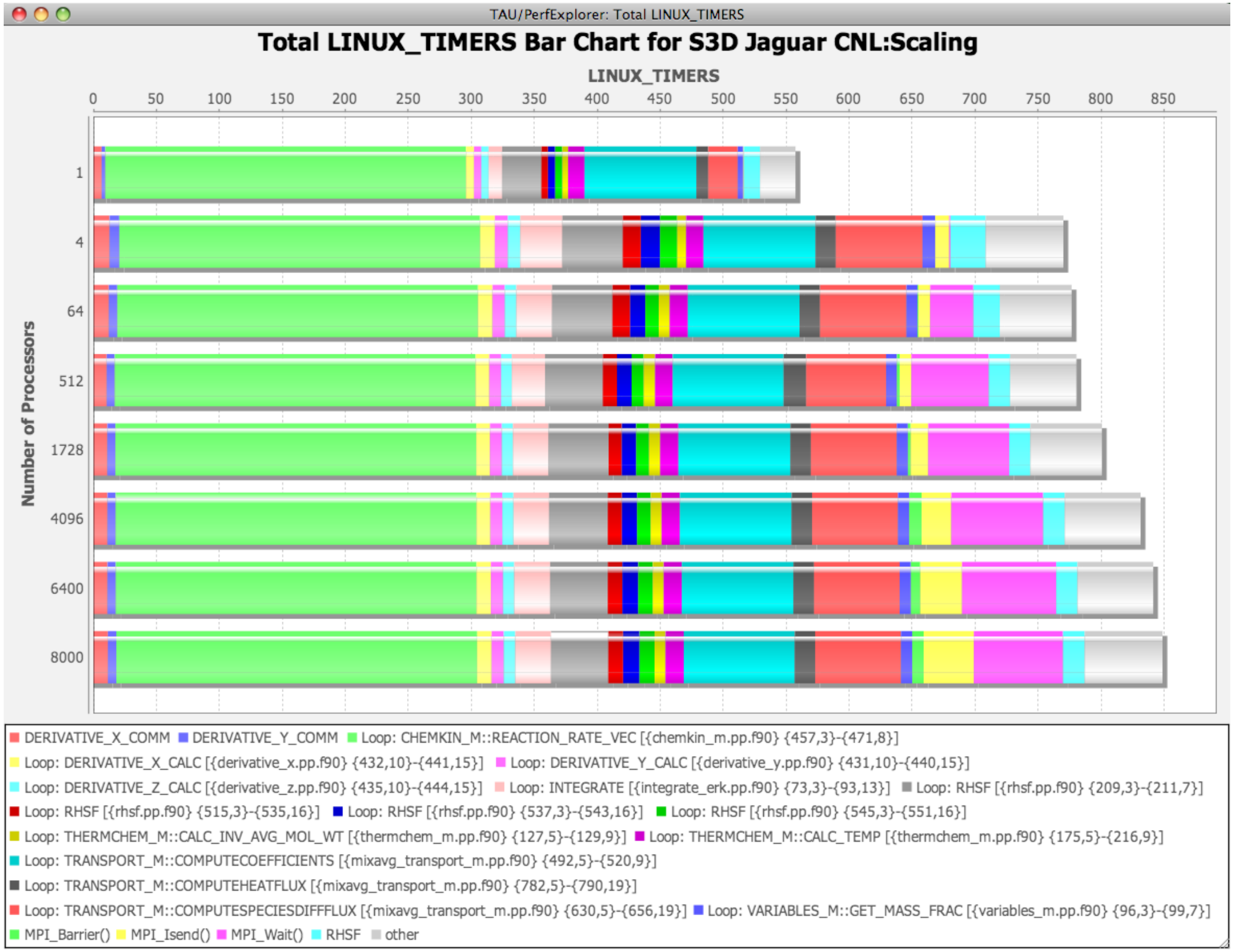
Usage Scenarios: Evaluate Scalability

- Goal: How does my application scale? What bottlenecks at what CPU counts?
- Load profiles in PerfDMF database and examine with PerfExplorer

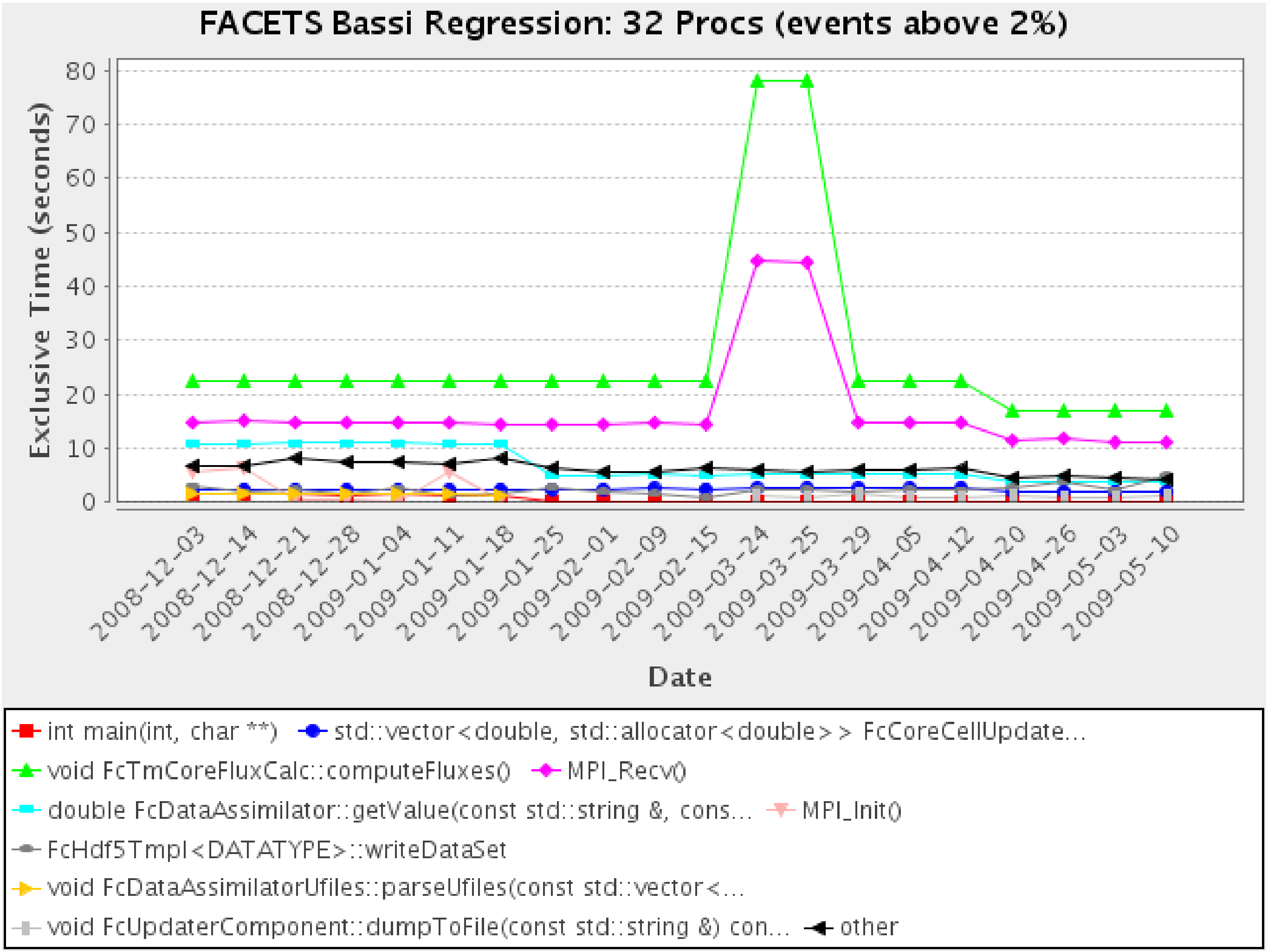




Usage Scenarios: Evaluate Scalability



Performance Regression Testing



Evaluate Scalability using PerfExplorer Charts

```
% export TAU_MAKEFILE=<taudir>/<arch>
                 /lib/Makefile.tau-mpi-pdt
% export PATH=<taudir>/<arch>/bin:$PATH
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)
% qsub  run1p.job
% paraprof --pack 1p.ppk
% qsub  run2p.job …
% paraprof --pack 2p.ppk … and so on.
On your client:
% taudb_configure -create-default
% perfexplorer_configure
(Yes to load schema, defaults)
% paraprof
(load each trial: DB -> Add Trial -> Type (Paraprof Packed
Profile) -> OK, OR use taudb_loadtrial on the commandline)
% perfexplorer
(Charts -> Speedup)
```

- TAU Portal
  - Support collaborative performance study
- Kernel-level system measurements (KTAU)
  - Application to OS noise analysis and I/O system analysis
- TAU performance monitoring
  - TAUoverSupermon and TAUoverMRNet
- PerfExplorer integration and expert-based analysis
  - OpenUH compiler optimizations
  - Computational quality of service in CCA
- Eclipse CDT and PTP integration
- Performance tools integration (NSF POINT project)

Support Acknowledgements

- US Department of Energy (DOE)
  - Office of Science contracts
  - SciDAC, LBL contracts
  - LLNL-LANL-SNL ASC/NNSA contract
  - Battelle, PNNL contract
  - ANL, ORNL contract
- Department of Defense (DoD)
  - PETTT, HPTi
- National Science Foundation (NSF)
  - SDCI, SI-2
- University of Oregon
- ParaTools, Inc.
- University of Tennessee, Knoxville
  - Dr. Shirley Moore
- T.U. Dresden, GWT
  - Dr. Wolfgang Nagel and Dr. Andreas Knupfer
- Research Centre Juelich
  - Dr. Bernd Mohr, Dr. Felix Wolf
    Dr. Markus Geimer, Dr. Brian Wylie

For more information

- TAU Website:
  http://tau.uoregon.edu
  - Software
  - Release notes
  - Documentation